mikroElektronika
Development tools - Books - Compilers
www.mikroe.com

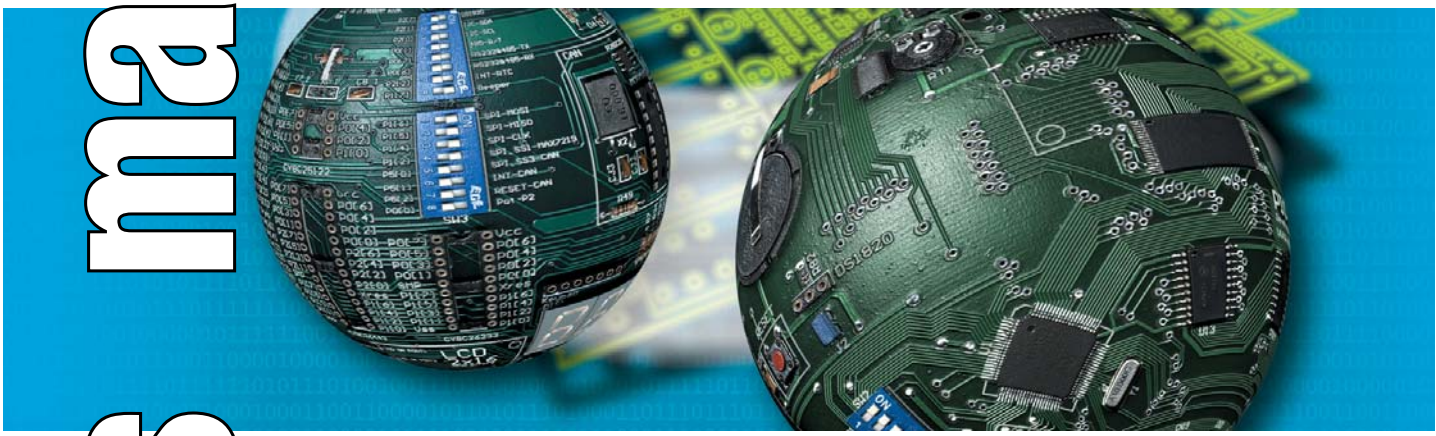
User's manual

BASIC Compiler for Microchip dsPIC30/33 and PIC24 Microcontrollers

mikroBasic

for dsPIC30/33 and PIC24

Making it simple



mikro ICD IN-CIRCUIT DEBUGGER

SUPPORTED from V5.0

Develop your applications quickly and easily with the world's most advanced BASIC compiler for dsPIC30/33 and PIC24 Microcontrollers (all families).

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, mikroBasic makes fast and reliable tool, which can satisfy the needs of experienced engineers and beginners alike.

Reader's note

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

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

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

This manual covers mikroBasic for dsPIC30/33 and PIC24 version 5.0 and the related topics. New 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 support@mikroe.com. Please include the following information in your bug report:

- Your operating system
- Version of mikroBasic
- 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
Support: support@mikroe.com

mikroBASIC User's manual

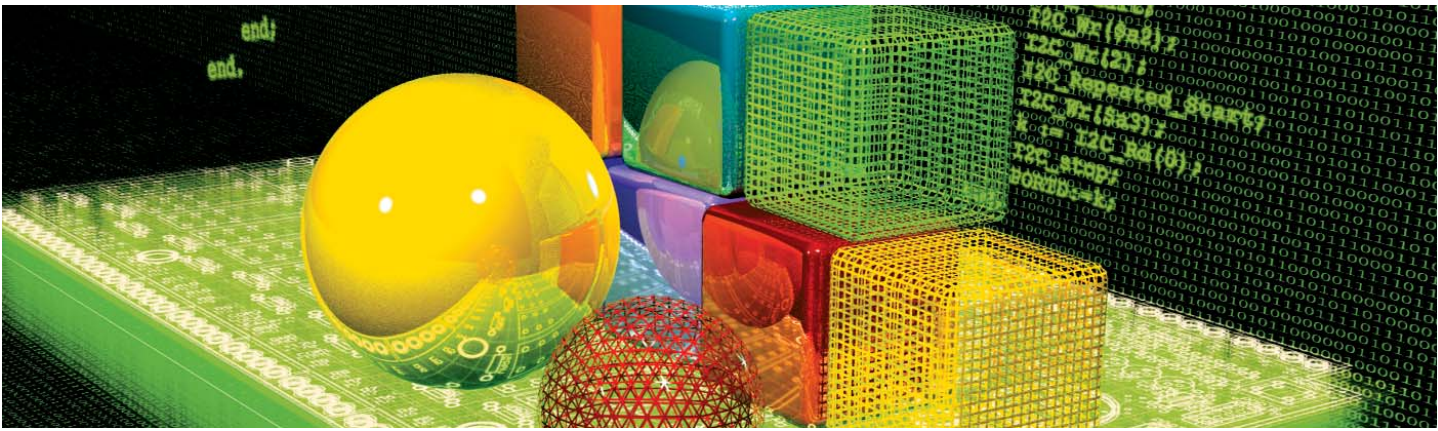
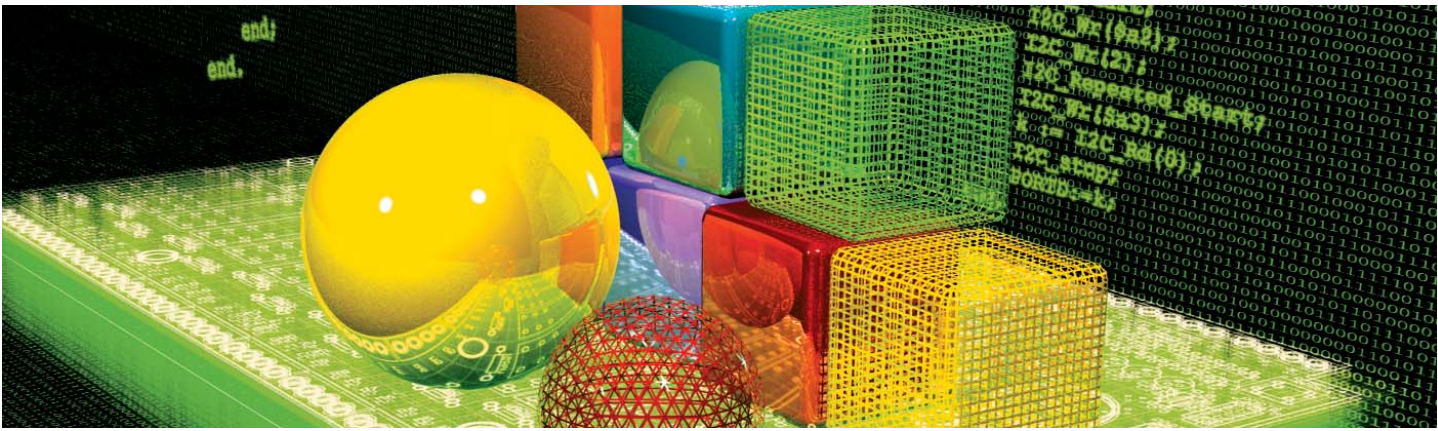


Table of Contents

CHAPTER 1

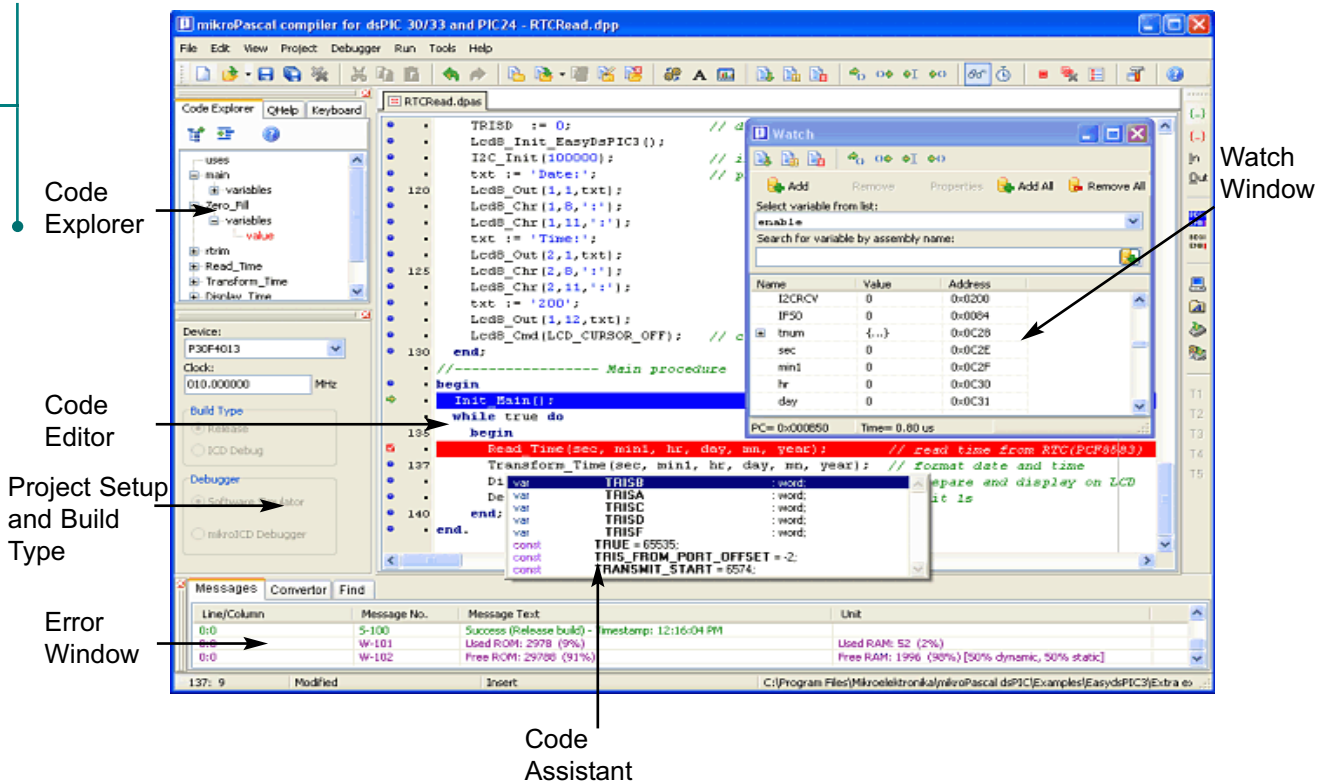


mikroBasic IDE

QUICK OVERVIEW

mikroBasic is a powerful, feature rich development tool for dsPIC30/33 and PIC24 micros. It has been designed to provide the customer with the easiest possible solution for developing applications for embedded systems, without compromising performance or control.

Highly advanced IDE, broad set of hardware libraries, comprehensive documentation and plenty of ready to run examples should be more than enough to get you started in programming dsPIC30/33 and PIC24 microcontrollers.



mikroBasic for dsPIC30/33 and PIC24 is an user-friendly and intuitive environment:

- The Code Editor features adjustable Syntax Highlighting, Code Assistant, Parameters Assistant, Auto Correct for common typos and Code Templates (Auto Complete).
- The Code Explorer (with Keyboard shortcut browser and Quick Help browser) is at your disposal for easier project management.
- The Error Window displays all errors detected during compiling and linking.
- The source-level Software Simulator lets you debug executable logic step-by-step by watching the program flow.
- The New Project Wizard is used to create a project in a fast, reliable, and easy way.
- Help files are syntax and context sensitive.
- Like in any modern Windows application, you may customize the layout of mikroBasic for dsPIC30/33 and PIC24 to suit your needs best.

CODE EDITOR

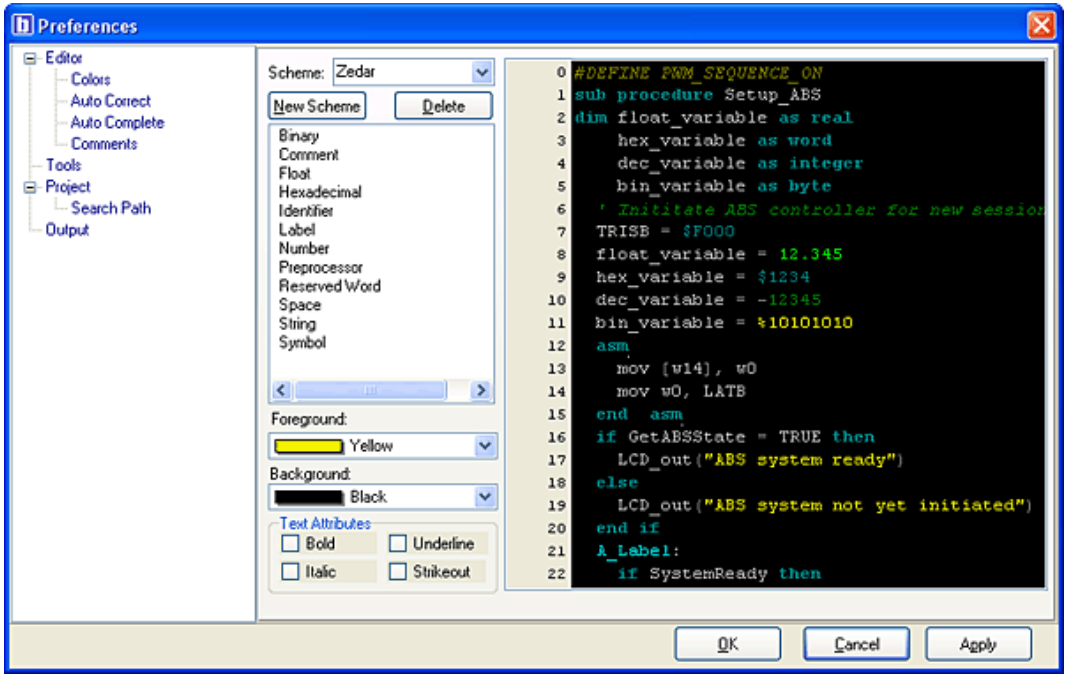
The Code Editor is an advanced text editor fashioned to satisfy needs of professionals. General code editing is the same as working with any standard text-editor, including familiar Copy, Paste and Undo actions, common for Windows environment.

Advanced Editor Features:

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

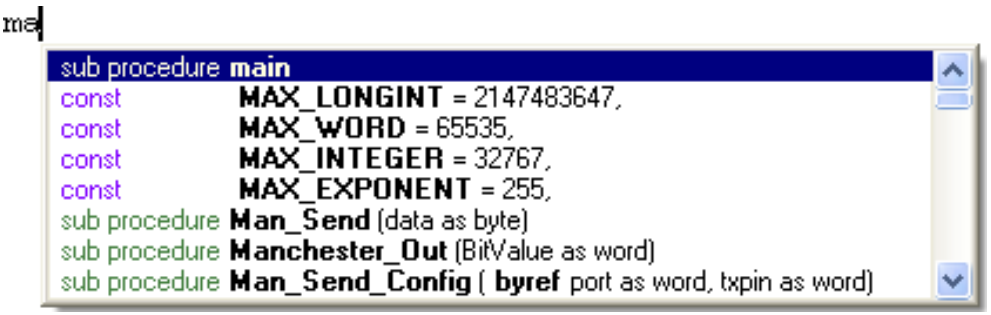
Tools Icon.

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



Code Assistant [CTRL+SPACE]

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

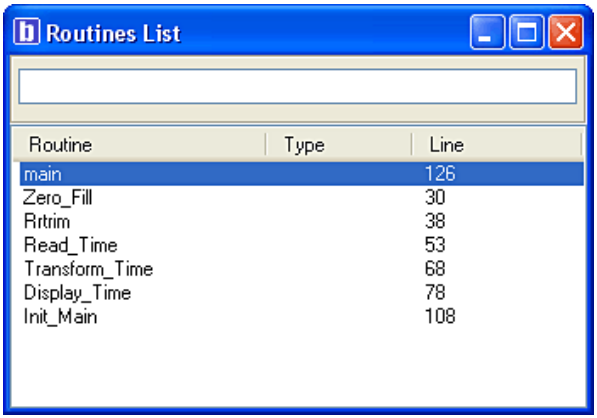


Parameter Assistant [CTRL+SHIFT+SPACE]

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

Routines List

If you press Ctrl+L, the Routines List Window will appear. All valid functions and procedures will be listed in this window (see the image below). Select the routine and double-click it or press Enter to move to the definition of the selected routine.



Code Templates [CTR+J]

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


Tools Icon.

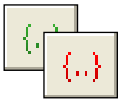
You can add your own templates to the list. Select **Tools › Preferences** from the drop-down menu, or click the Preferences Icon and then select the Auto Complete Tab. Here you can enter the appropriate keyword, description and code of your template.

Auto Correct


Tools Icon.

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

Comment/Uncomment


Comment / Uncomment Icon.

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

Bookmarks

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

Goto Line

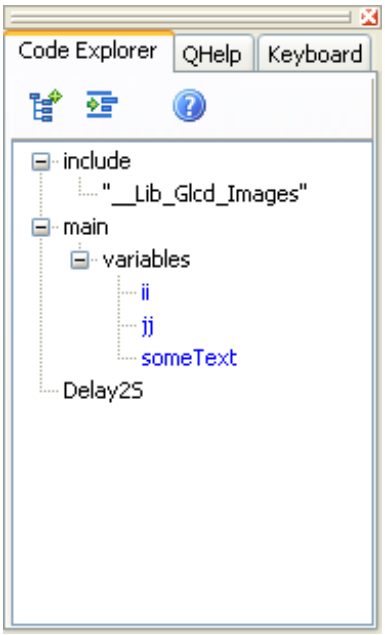
The Goto Line option makes navigation through a large code easier.
Use the shortcut Ctrl+G to activate this option.

CODE EXPLORER

The Code Explorer is placed to the left of the main window by default and gives clear view of each item declared inside the source code. You can jump to a declaration of any item by double clicking it. To expand/collapse the treeview in the Code Explorer, use the icon .

Collapse/Expand All Icon.

Also, there are two more tab windows available in the Code Explorer. QHelp Tab lists all available built-in and library functions, for a quick reference. Double-clicking a routine in the QHelp view opens the relevant Help topic. The Keyboard Tab lists all available keyboard shortcuts in mikroBasic for dsPIC30/33 and PIC24.



Software Simulator


Start Debugger.

The Source-level Software Simulator is an integral component of the mikroBasic for dsPIC30/33 and PIC24 environment. It is designed to simulate operations of the Microchip dsPIC30/33 and PIC24 MCUs and assist the users in debugging Basic code written for these devices.

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

Note: The Software Simulator simulates the program flow and execution of instruction lines, but it cannot fully emulate dsPIC30/33 and PIC24 device behavior, i.e. it doesn't update timers, interrupt flags, etc.

Name	Description	Function Key
`Start Debugger`	Start Software Simulator.	[F9]
`Run/Pause Debugger`	Run or pause Software Simulator.	[F6]
`Stop Debugger`	Stop Software Simulator.	[Ctrl+F2]
`Toggle Breakpoints`	Toggle breakpoint at the current cursor position. To view all breakpoints, select Run › View Breakpoints from the drop-down menu. Double clicking an item in the Breakpoints Window List locates the breakpoint.	[F5]
`Run to cursor`	Execute all instructions between the current instruction and cursor position.	[F4]
`Step Into`	Execute the current Basic (single or multi–cycle) instruction, then halt. If the instruction is a routine call, enter the routine and halt at the first instruction following the call.	[F7]
`Step Over`	Execute the current Basic (single or multi–cycle) instruction, then halt.	[F8]
`Step Out`	Execute all remaining instructions in the current routine, return and then halt.	[Ctrl+F8]

Watch Window

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

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

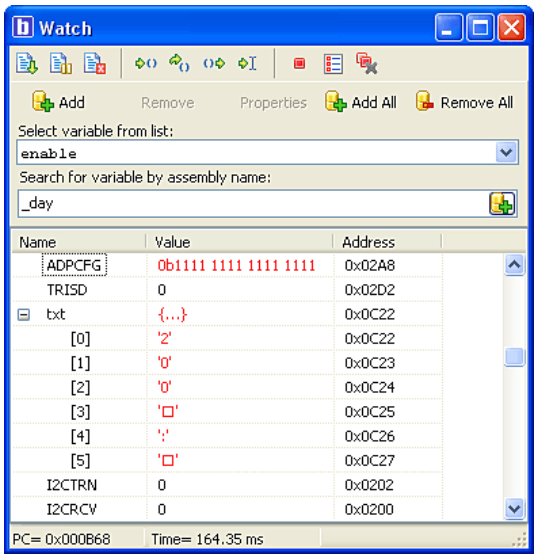
There are two ways of adding variable/register to the watch list:

ADD Button

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

Remove Button

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



Add All Button

Add All Button adds all variables.

Remove All Button

Remove All Button removes all variables.

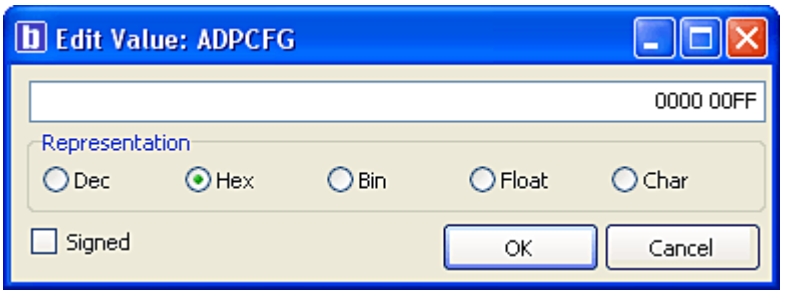
You can also expand/collapse complex variables, i.e. struct type variables, strings...

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

Properties Button

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

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



Stopwatch Window

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

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

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

RAM Window

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

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

RAM

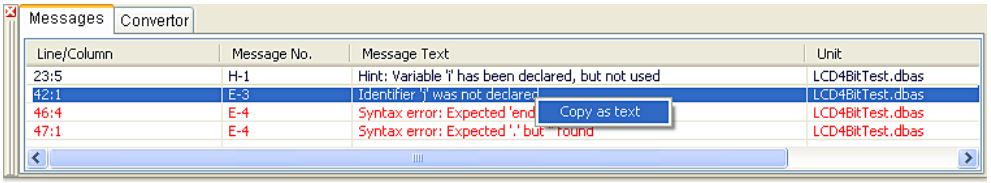
RAM

	00	02	04	06	08	0A	0C	0E	ASCII
0000	0BFE	0000	0000	0000	0000	0000	0000	0000	þ<VT> .. ..
0010	0000	0000	0000	0000	0000	0000	0000	0800	.. ..<BS>
0020	0BFE	0000	0000	0000	0000	0000	0000	0B66	þ<VT> .. .. f<VT>
0030	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
0040	0000	0000	0800	0000	0000	0000	0000	0000	.. .. .<BS> .. ..
0050	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
0060	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
0070	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
0080	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
0090	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
00A0	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
00B0	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
00C0	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
00D0	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
00E0	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
00F0	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
0100	0000	0000	0000	0000	0000	0000	0000	0000	.. ..

ERROR WINDOW

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

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



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

Right-Click the message line to enable copying the message text to Clipboard.

For more information on errors recognized by the compiler consult the Error Messages.

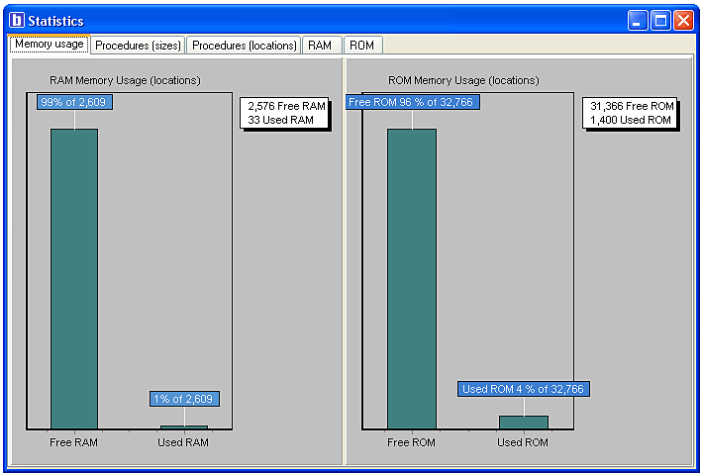
STATISTICS

Statistics Icon.

After successful compilation, you can review statistics of your code. Select **View › View Statistics** from the drop-down menu, or click the Statistics Icon . There are five tab windows:

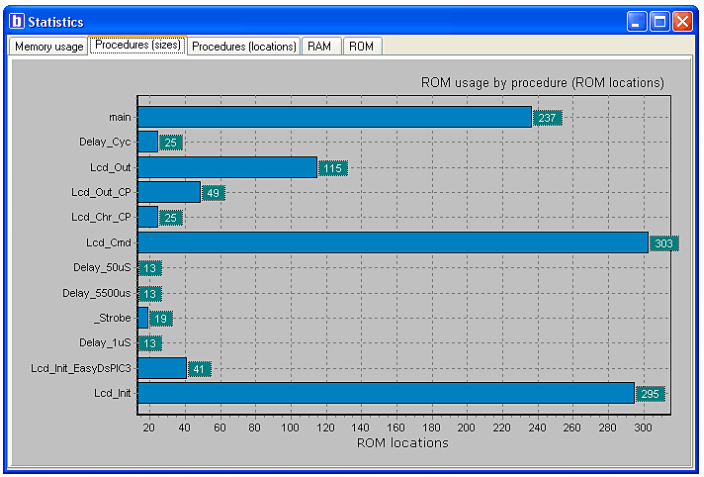
Memory Usage Window

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



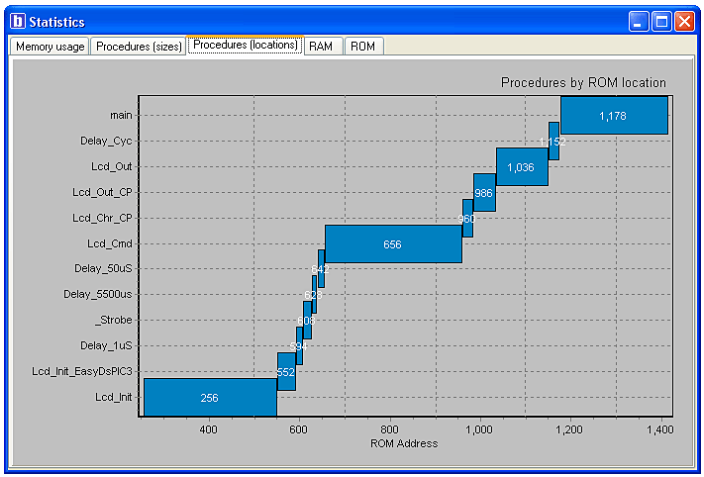
Procedures (Graph) Window

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



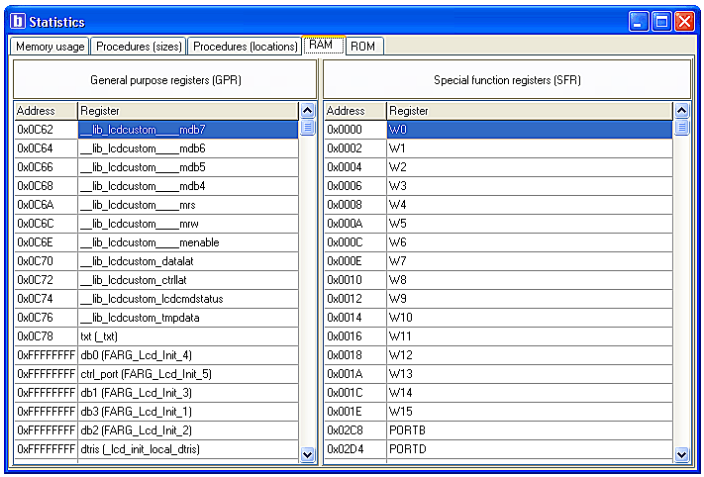
Procedures (Locations) Window

Displays how procedures and functions are distributed in microcontroller's memory.



RAM Window

Summarizes all GPRs and SFRs and their addresses. It also displays symbolic names of variables and their addresses.

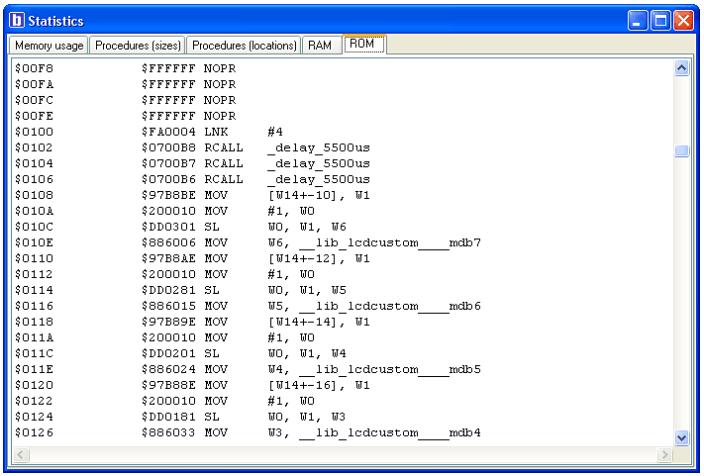


Address	Register
0x0C62	__lib_lcdcustom____mdb7
0x0C64	__lib_lcdcustom____mdb6
0x0C66	__lib_lcdcustom____mdb5
0x0C68	__lib_lcdcustom____mdb4
0x0C6A	__lib_lcdcustom____mrs
0x0C6C	__lib_lcdcustom____mrw
0x0C6E	__lib_lcdcustom____menable
0x0C70	__lib_lcdcustom_datalat
0x0C72	__lib_lcdcustom_ctrllat
0x0C74	__lib_lcdcustom_lcdcmdstatus
0x0C76	__lib_lcdcustom_tmpdata
0x0C78	txt (_txt)
0xFFFFFFFF	db0 (FARG_Lcd_Init_4)
0xFFFFFFFF	ctrl_port (FARG_Lcd_Init_5)
0xFFFFFFFF	db1 (FARG_Lcd_Init_3)
0xFFFFFFFF	db3 (FARG_Lcd_Init_1)
0xFFFFFFFF	db2 (FARG_Lcd_Init_2)
0xFFFFFFFF	dtris (_lcd_init_local_dtris)

Address	Register
0x0000	W0
0x0002	W1
0x0004	W2
0x0006	W3
0x0008	W4
0x000A	W5
0x000C	W6
0x000E	W7
0x0010	W8
0x0012	W9
0x0014	W10
0x0016	W11
0x0018	W12
0x001A	W13
0x001C	W14
0x001E	W15
0x02C8	PORTB
0x02D4	PORTD

ROM Window

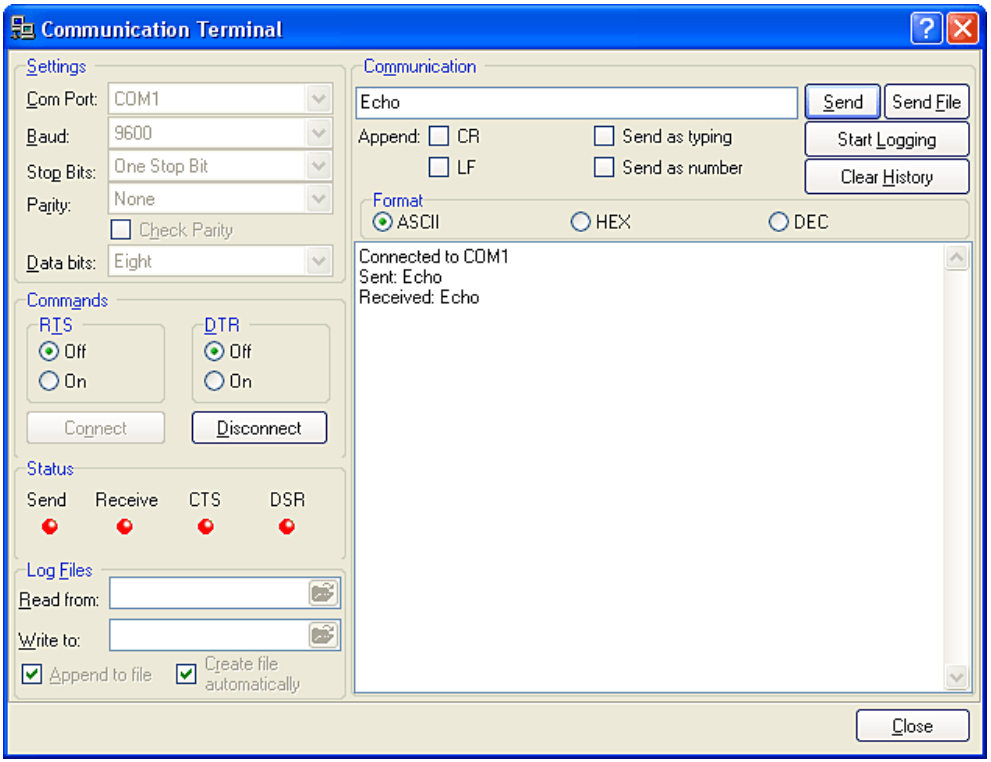
Lists op-codes and their addresses in form of a human readable hex code.




USART Terminal Icon

USART Terminal

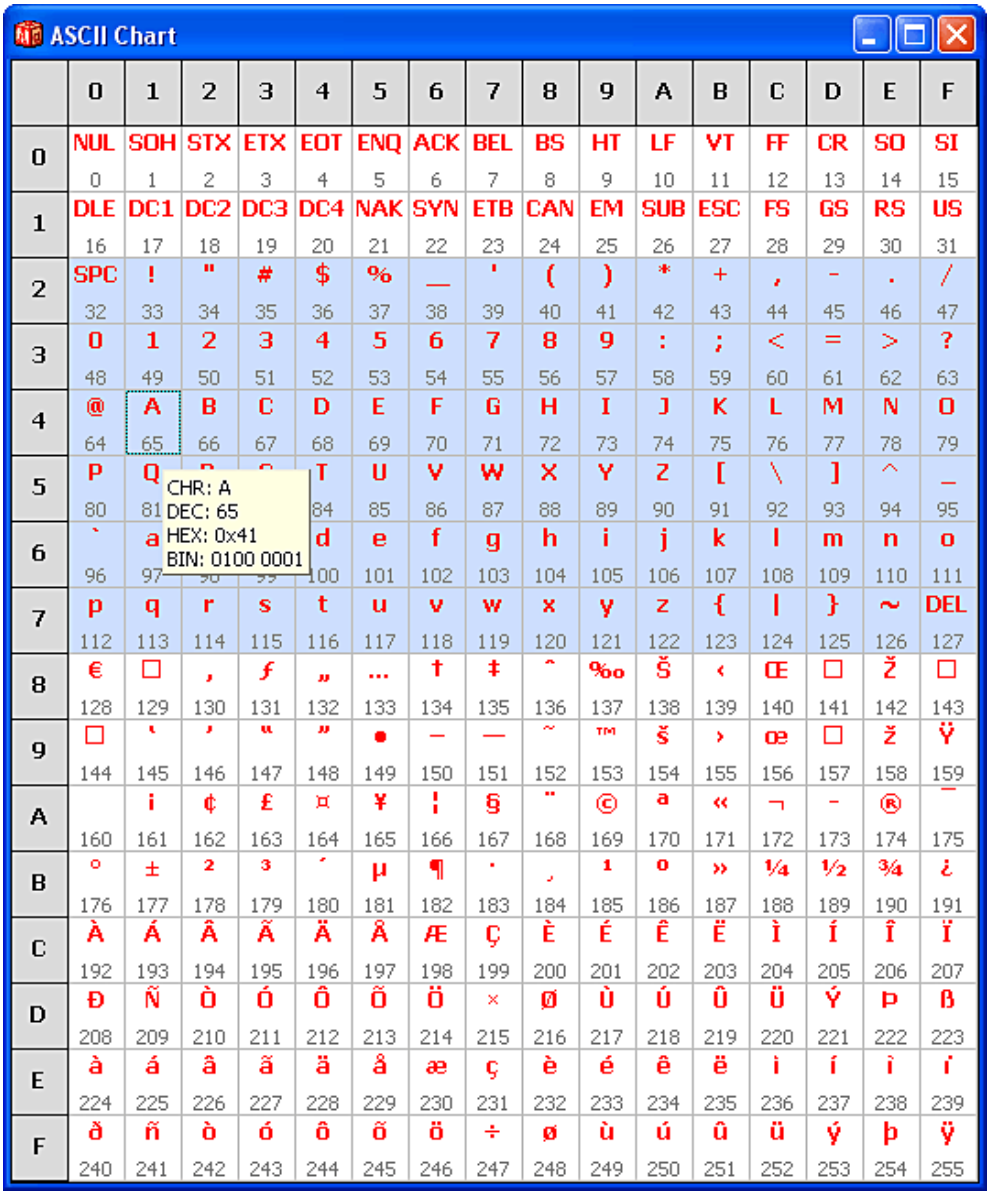
mikroBasic for dsPIC30/33 and PIC24 includes the USART communication terminal for RS232 communication. You can launch it from the drop-down menu **Tools › USART Terminal** or by clicking the Terminal Icon.



ASCII Chart Icon

ASCII Chart

The ASCII Chart is a handy tool, particularly useful when working with LCD display. You can launch it from the drop-down menu **Tools › Ascii chart** or by clicking the View ASCII Chart Icon.



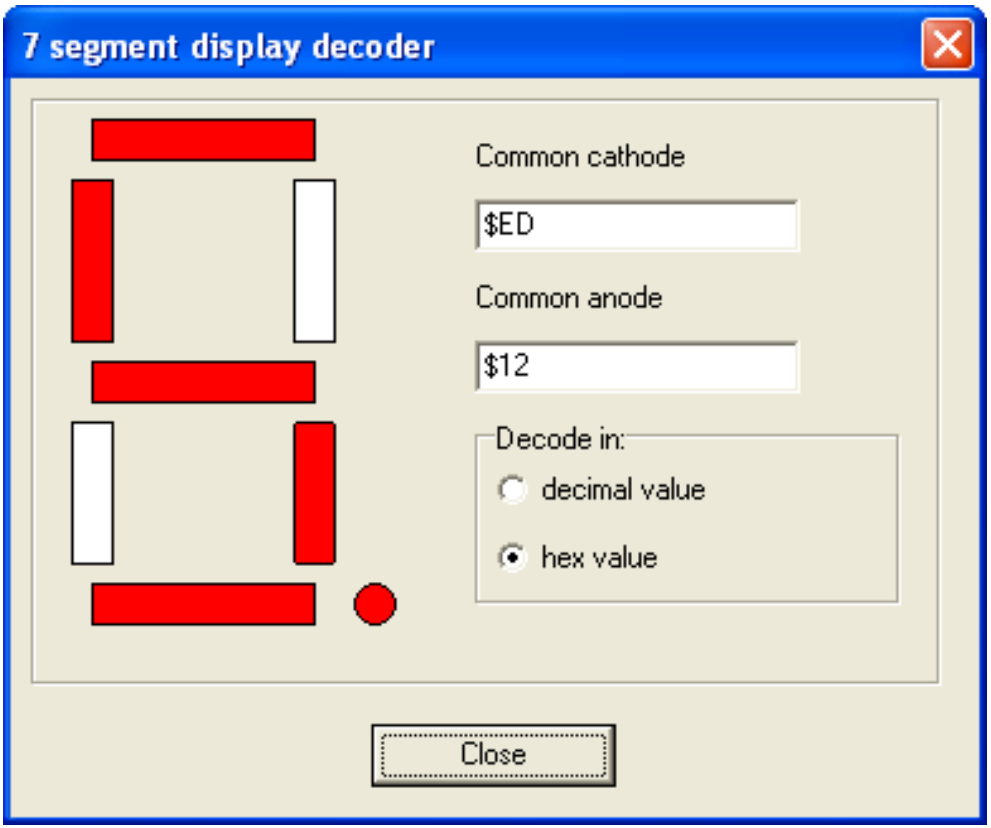
ASCII Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL 0	SOH 1	STX 2	ETX 3	EOT 4	ENQ 5	ACK 6	BEL 7	BS 8	HT 9	LF 10	VT 11	FF 12	CR 13	SO 14	SI 15
1	DLE 16	DC1 17	DC2 18	DC3 19	DC4 20	NAK 21	SYN 22	ETB 23	CAN 24	EM 25	SUB 26	ESC 27	FS 28	GS 29	RS 30	US 31
2	SPC 32	! 33	" 34	# 35	$ 36	% 37	_ 38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
3	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
4	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
5	P 80	Q 81	[illegible]	[illegible]	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
6	` 96	a 97	[illegible]	[illegible]	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	DEL 127
8	€ 128	□ 129	‚ 130	ƒ 131	„ 132	… 133	† 134	‡ 135	ˆ 136	‰ 137	Š 138	‹ 139	Œ 140	□ 141	Ž 142	□ 143
9	□ 144	‘ 145	’ 146	“ 147	” 148	• 149	– 150	— 151	˜ 152	™ 153	š 154	› 155	œ 156	□ 157	ž 158	Ÿ 159
A	160	¡ 161	¢ 162	£ 163	¤ 164	¥ 165	¦ 166	§ 167	¨ 168	© 169	ª 170	« 171	¬ 172	- 173	® 174	¯ 175
B	° 176	± 177	² 178	³ 179	´ 180	µ 181	¶ 182	· 183	¸ 184	¹ 185	º 186	» 187	¼ 188	½ 189	¾ 190	¿ 191
C	À 192	Á 193	Â 194	Ã 195	Ä 196	Å 197	Æ 198	Ç 199	È 200	É 201	Ê 202	Ë 203	Ì 204	Í 205	Î 206	Ï 207
D	Ð 208	Ñ 209	Ò 210	Ó 211	Ô 212	Õ 213	Ö 214	× 215	Ø 216	Ù 217	Ú 218	Û 219	Ü 220	Ý 221	Þ 222	ß 223
E	à 224	á 225	â 226	ã 227	ä 228	å 229	æ 230	ç 231	è 232	é 233	ê 234	ë 235	ì 236	í 237	î 238	ï 239
F	ð 240	ñ 241	ò 242	ó 243	ô 244	õ 245	ö 246	÷ 247	ø 248	ù 249	ú 250	û 251	ü 252	ý 253	þ 254	ÿ 255

CHR: A
DEC: 65
HEX: 0x41
BIN: 0100 0001

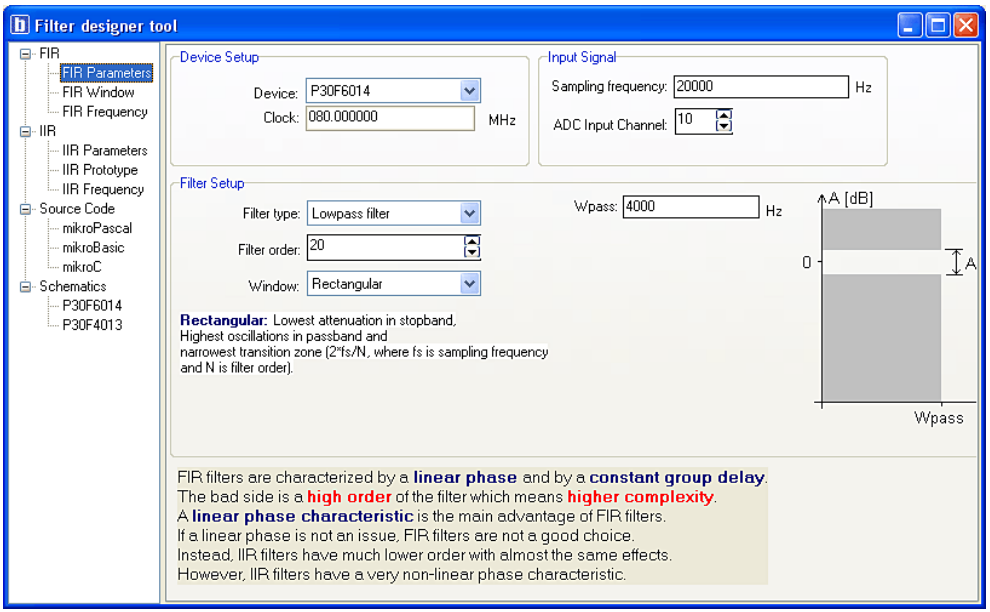
7 Segment Display Decoder

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



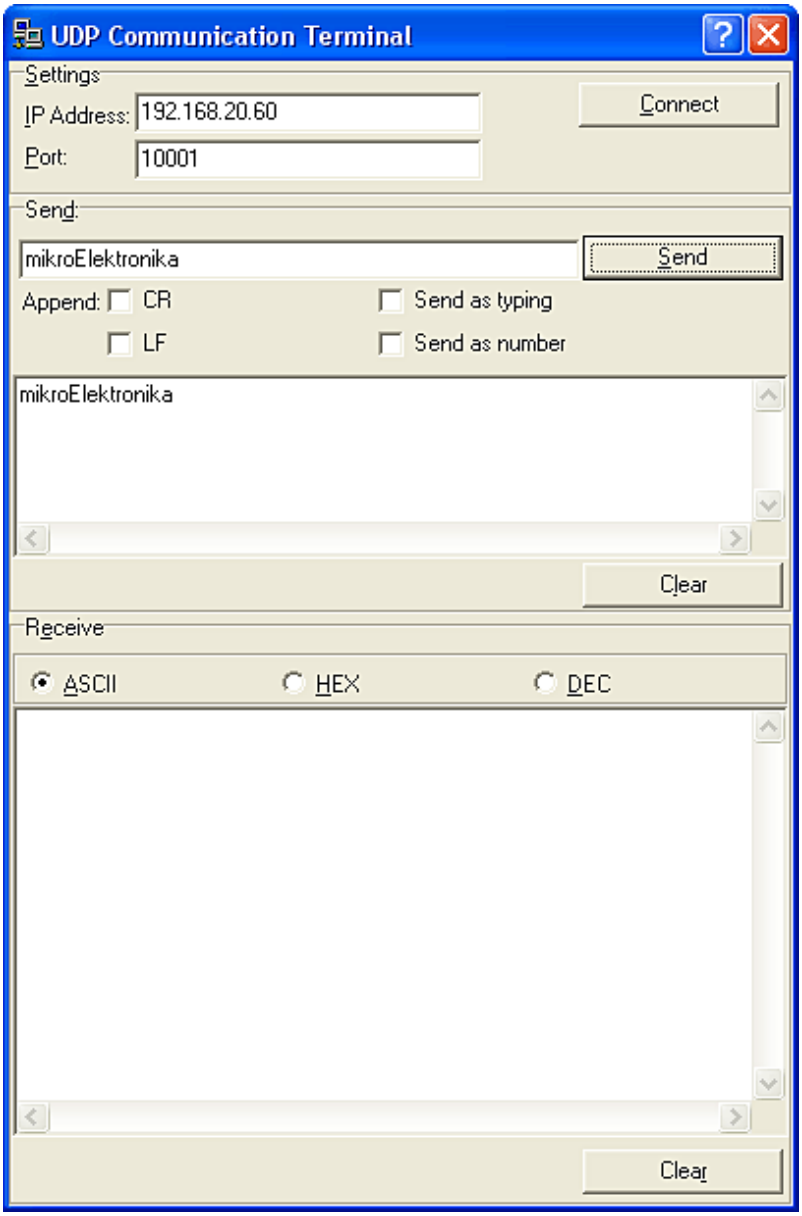
Filter Designer

The Filter designer is a tool for designing FIR and IIR filters. It has an user-friendly visual interface for setting filter parameters. Filter designer output is mikroBasic for dsPIC30/33 and PIC24 compatible code. You can launch it from the drop-down menu **Tools › Filter Designer Tool**.



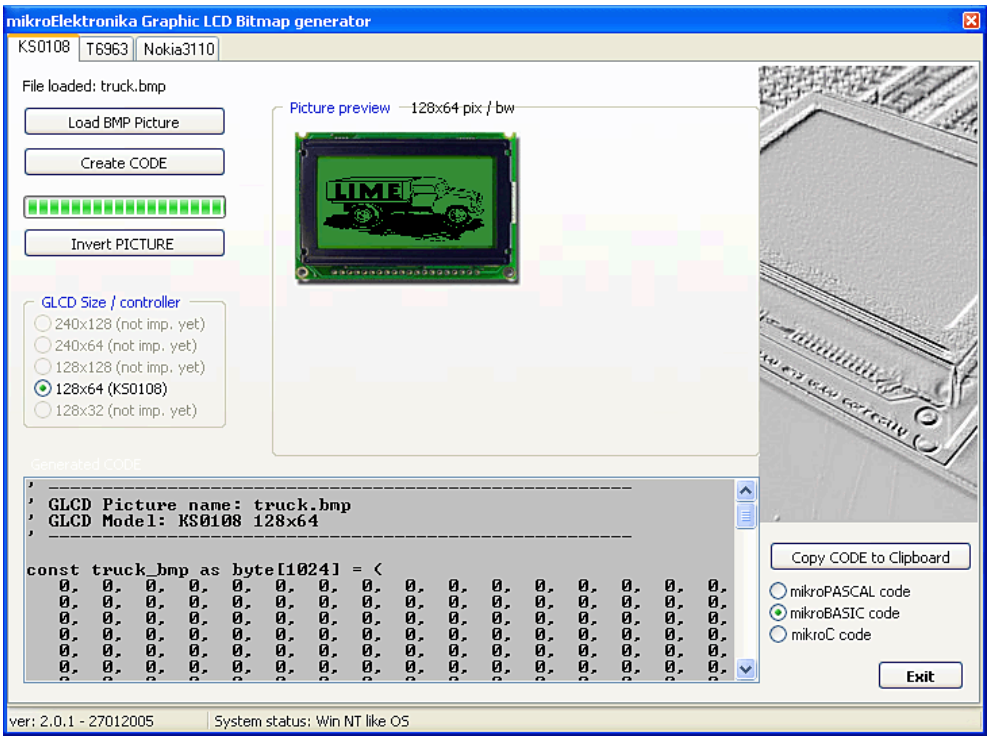
UDP Terminal

mikroBasic for dsPIC30/33 and PIC24 includes the UDP Terminal. You can launch it from the drop-down menu **Tools › UDP Terminal**.



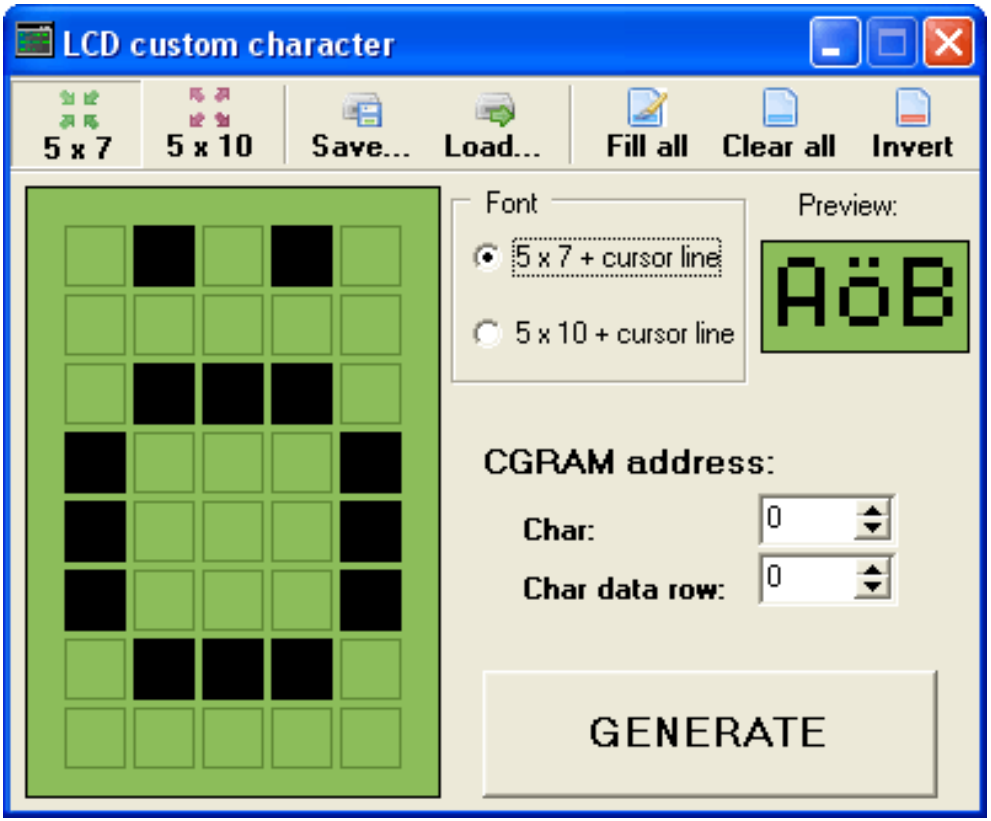
Graphic LCD Bitmap Editor

mikroBasic for dsPIC30/33 and PIC24 includes the Graphic LCD Bitmap Editor. Output is mikroBasic for dsPIC30/33 and PIC24 compatible code. You can launch it from the drop-down menu **Tools › GLCD Bitmap Editor**.



LCD Custom Character

mikroBasic for dsPIC30/33 and PIC24 includes the LCD Custom Character. Output is mikroBasic for dsPIC30/33 and PIC24 compatible code. You can launch it from the drop-down menu **Tools › LCD Custom Character**.



KEYBOARD SHORTCUTS

Below is a complete list of keyboard shortcuts available in mikroBasic for dsPIC30/33 and PIC24 IDE. You can also view keyboard shortcuts in the Code Explorer window, tab Keyboard.

IDE Shortcuts

F1	Help
Ctrl+N	New Unit
Ctrl+O	Open
Ctrl+F9	Compile
Shift+F9	Compile All
F11	Program
Ctrl+F11	Compile and Program
F12	Preferences
Ctrl+Shift+E	Edit Project
Shift+F4	View Breakpoints
Ctrl+Shift+F5	Clear Breakpoints

Basic Editor shortcuts

F3	Find Next
Alt+F3	Find in Files
Ctrl+A	Select All
Ctrl+C	Copy
Ctrl+F	Find
Ctrl+H	Replace
Ctrl+P	Print
Ctrl+S	Save Unit
Ctrl+Shift+S	Save All
Ctrl+V	Paste
Ctrl+X	Cut
Ctrl+Y	Delete Line
Ctrl+Z	Undo
Ctrl+Shift+Z	Redo

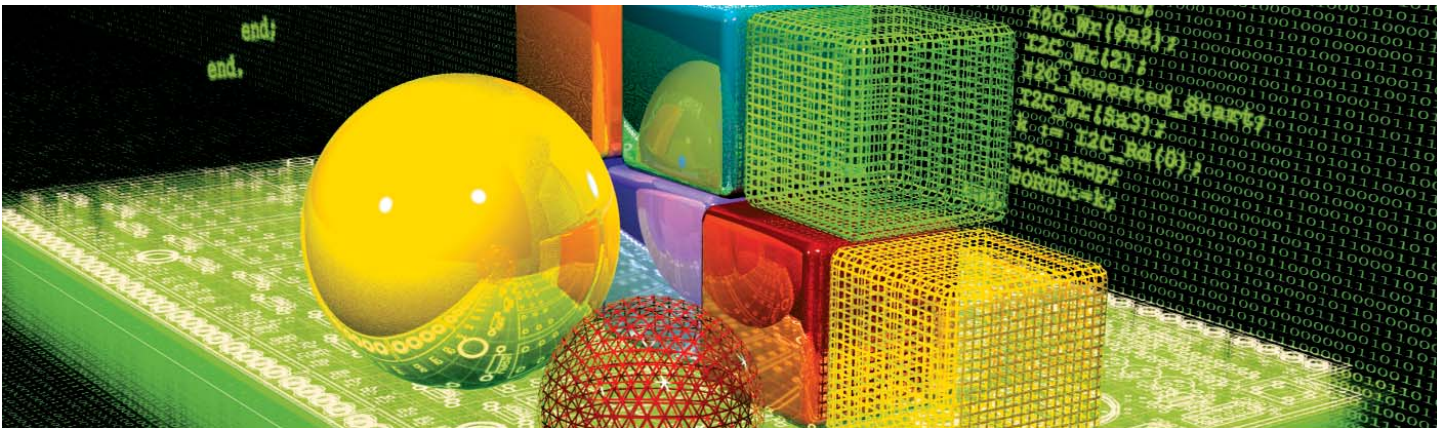
Advanced Editor shortcuts

Ctrl+Space	Code Assistant
Ctrl+Shift+Space	Parameters Assistant
Ctrl+D Find	Declaration
Ctrl+E	Incremental Search
Ctrl+G	Goto Line
Ctrl+J	Insert Code Template
Ctrl+L	Procedures List
Ctrl+/	Toggle Line Comment
Ctrl+Shift+.	Comment Code
Ctrl+Shift+,	Uncomment Code
Ctrl+number	Goto Bookmark
Ctrl+Shift+number	Set Bookmark
Ctrl+Shift+I	Indent Selection
Ctrl+Shift+U	Unindent Selection
TAB	Indent Selection
Shift+TAB	Unindent Selection
Alt+Select	Select Columns
Ctrl+Alt+Select	Select Columns

Debugger Shortcuts

F4	Run to Cursor
F5	Toggle Breakpoint
F6	Run/Pause Debugger
F7	Step Into
F8	Step Over
F9	Start Debugger
Ctrl+F2	Stop Debugger
Ctrl+F5	Add to Watch
Ctrl+F8	Step Out

CHAPTER 2



Building Applications

Creating applications in mikroBasic 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.

mikroBasic allows you to distribute your projects in as many units as you find appropriate. You can then share your mikroCompiled Libraries (`.mcl` files) with other developers without disclosing the source code.

PROJECTS

mikroBasic for dsPIC30/33 and PIC24 organizes applications into projects, consisting of a single project file (extension .dbp) and one or more .dbas source files. You can compile source files only if they are a part of the project.

The project file contains the following information:

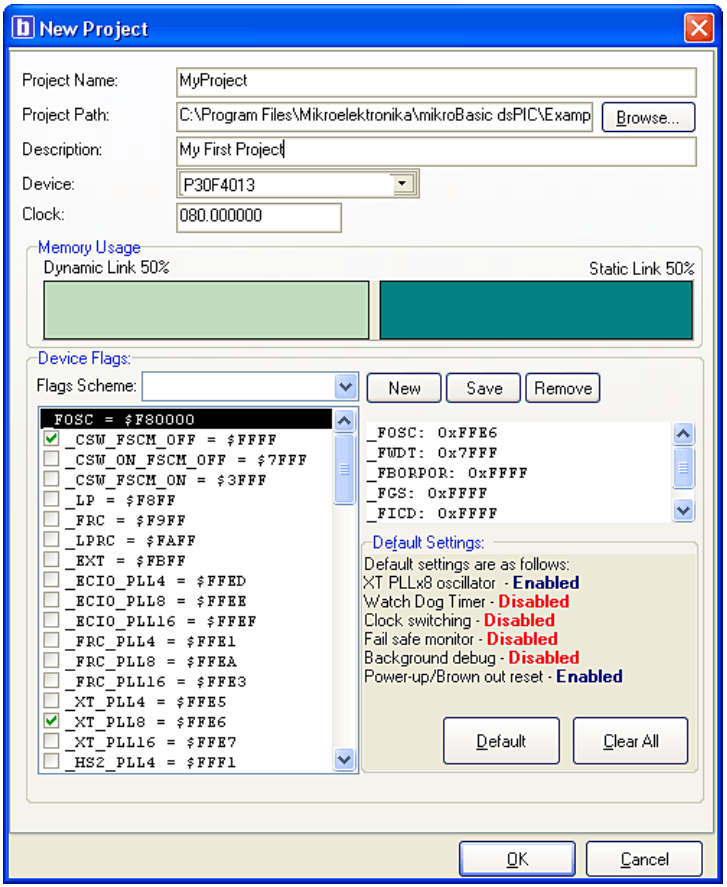
- project name and optional description,
- target device,
- device flags (config word),
- device clock,
- list of the project source files with paths.

New Project


New Project.

The easiest way to create a project is by means of the New Project Wizard, drop-down menu **Project › New Project** or by clicking the New Project Icon . Just fill the dialog with desired values (project name and description, location, device, clock, config word) and mikroBasic for dsPIC30/33 and PIC24 will create an appropriate project file.

There is also a slider for dsPIC30/33 and PIC24 data memory usage. You can specify static and dynamic memory consumption. Static memory is assigned for global variables and dynamic memory is assigned for local variables. Neither static nor dynamic memory can occupy more than 80 % of total data memory space.



Editing Project


Edit Project.

Afterwards, you can change project settings from the drop-down menu **Project › Edit Project** or by clicking the Edit Project Icon . 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 (extension `.dbp`) is stored.

SOURCE FILES

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

Search Paths

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

When including source files with the uses clause, mikroBasic for dsPIC30/33 and PIC24 will look for the file in the following locations according to this particular order:

1. mikroBasic for dsPIC30/33 and PIC24 installation folder › "defs" folder
2. mikroBasic for dsPIC30/33 and PIC24 installation folder › "uses" folder
3. the project folder (folder which contains the project file `.dbp`)
4. your custom search paths

Managing Source Files

Creating new source file

To create a new source file, do the following:


New File.

1. Select **File › New** from the drop-down menu, or press Ctrl+N, or click the New File Icon .
2. A new tab will be opened. It is a new source file. Select **File › Save As** from the drop-down menu and name it as you wish.

If you use the New Project Wizard, an empty source file named after the project with extension .dbas, is created automatically. mikroBasic for dsPIC30/33 and PIC24 does not require you to have a source file named the same as the project, it's just a matter of convenience.

Opening an Existing File


Open File.

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

Printing an Open File


Print File.

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

Saving File


Save File.

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

Saving File Under a Different Name


Save File As.

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

Closing a File


Close File.

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

COMPILATION


Build Icon
Build Project Icon

Compilation

When you have created a project and written a source code, it's time to compile it. Select **Project › Build** from the drop-down menu, or click the Build Project Icon from the Compiler Toolbar.

Progress bar will appear to inform you about the status of compiling. If there are some errors, you will be notified in the Error Window. If no errors are encountered, mikroBasic for dsPIC30/33 and PIC24 will generate output files.

Output Files

Upon successful compilation, mikroBasic for dsPIC30/33 and PIC24 will generate output files in the project folder (a folder which contains the project file .dbp). Output files are summarized below:

Intel HEX file (`.hex`)
Intel style hex records. Use this file to program dsPIC30/33 and PIC24 MCU.

Binary mikro Compiled Library (`.mcl`)
mikro Compiled Library. Binary distribution of application that can be included in other projects.

List File (`.lst`)
Overview of the dsPIC30/33 and PIC24 memory allotment: instruction addresses, registers, routines and labels.

Assembler File (`.asm`)
Human-readable assembly with symbolic names, extracted from the List File.


View Assembly Icon.

Assembly View

After compiling the program in mikroBasic for dsPIC30/33 and PIC24, you can click the View Assembly icon or select View › View Assembly from the drop-down menu to review the generated assembly code (`.asm` file) in a new tab window. Assembly is human-readable with symbolic names. All physical addresses and other information can be found in the Statistics Window or List File (`.lst`).

ERROR MESSAGES

Compiler Error Messages

Message	Message Number
Error: "%s" is not valid identifier	1
Error: Unknown type "%s"	2
Error: Identifier "%s" was not declared	3
Error: Expected "%s" but "%s" found	4
Error: Argument is out of range	5
Error: Syntax error in additive expression	6
Error: File "%s" not found	7
Error: Invalid command "%s"	8
Error: Not enough parameters	9
Error: Too many parameters	10
Error: Too many characters	11
Error: Actual and formal parameters must be identical	12
Error: Invalid ASM instruction: "%s"	13
Error: Identifier "%s" has been already declared	14
Error: Syntax error in multiplicative expression	15
Error: Definition file for "%s" is corrupted	16

Message	Message Number
Error: ORG directive is currently supported for interrupts only	17
Error: Not enough ROM	18
Error: Not enough RAM	19
Error: External procedure "%s" used in "%s" was not found	20
Error: Internal error: "%s"	21
Error: Unit cannot recursively use itself	22
Error: "%s" cannot be used out of loop	23
Error: Supplied and formal parameters do not match ("%s" to "%s")	24
Error: Constant cannot be assigned to	25
Error: Constant array must be declared as global	26
Error: Incompatible types ("%s" to "%s")	27
Error: Too many characters ("%s")	28
Error: Soft_Uart cannot be initialized with selected baud rate/device clock	29
Error: Main label cannot be used in modules	30
Error: Break/Continue cannot be used out of loop	31
Error: Preprocessor Error: "%s"	32
Error: Expression is too complicated	33

Message	Message Number
Error: Duplicated label "%s"	34
Error: Complex type cannot be declared here	35
Error: Record is empty	36
Error: Unknown type "%s"	37
Error: File not found "%s"	38
Error: Constant parameter cannot be passed by reference	39
Error: Pointer parameter cannot be passed by reference	40
Error: Improper use of operator "%s"	41
Error: Exit cannot be called from the main block	42
Error: Array parameter must be passed by reference	43
Error: Error occured while compiling "%s"	44
Error: Recursive types are not allowed	45
Error: Adding strings is not allowed, use "strcat" procedure instead	46
Error: Cannot declare pointer to array, use pointer to structure which has array field	47
Error: Return value of the function "%s" is not defined	48
Error: Assignment to for loop variable is not allowed	49
Error: "%s" is allowed only in the main program	50

Message	Message Number
Error: Start address of "%s" has already been defined	51
Error: Simple constant cannot have a fixed address	52
Error: Invalid date/time format	53

Hint and Warning Messages

Message	Message Number
Hint: Variable "%s" has been declared, but not used	1
Warning: Variable "%s" is not initialized	2
Warning: Return value of the function "%s" is not defined	3
Hint: Constant "%s" has been declared, but not used	4
Warning: Identifier "%s" overrides declaration in unit "%s"	5
Warning: Generated baud rate is "%s" bps (error = "%s" percent)	6
Warning: Result size may exceed destination array size	7
Warning: Infinite loop	8
Warning: Implicit typecast performed from "%s" to "%s"	9
Hint: Unit "%s" has been recompiled	10
Hint: Variable "%s" has been eliminated by optimizer	11
Warning: Implicit typecast of integral value to pointer	12

Message	Message Number
Warning: Library "%s" was not found in search path	13
Warning: Interrupt context saving has been turned off	14

Linker Error Messages

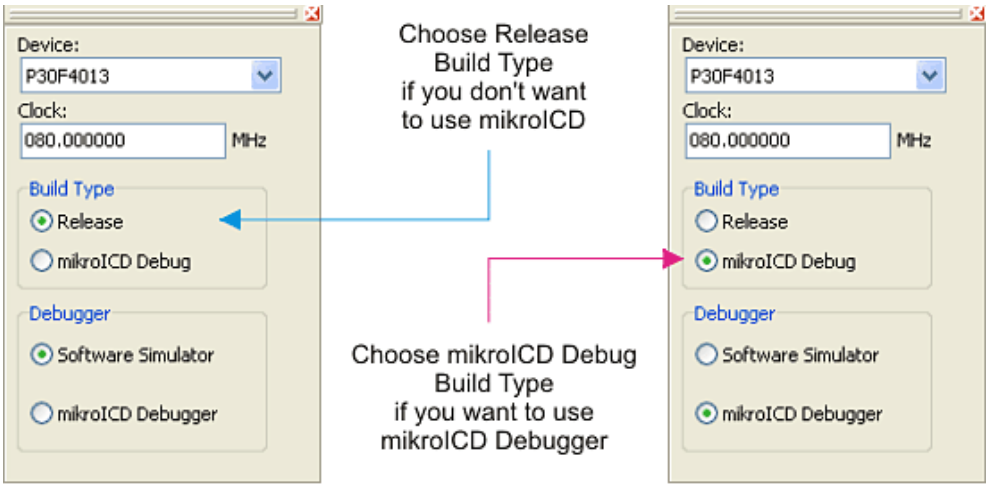
Message	Message Number
Linker error: "%s" "%s"	1
Warning: Variable "%s" is not initialized	2
Warning: Return value of the function "%s" is not defined	3
Hint: Constant "%s" has been declared, but not used	4
Warning: Identifier "%s" overrides declaration in unit "%s"	5
Linker error: Constant "%s" was not found	6
Linker error: Address of the routine has already been defined	7

mikroICD (In-Circuit Debugger)

The mikroICD is a highly effective tool for **Real-Time debugging** on hardware level. The mikroICD debugger enables you to execute the mikroBasic for dsPIC30/33 and PIC24 program on the host dsPIC30/33 or PIC24 microcontroller and view variable values, Special Function Registers (SFRs), RAM, CODE and EEPROM along with the mikroICD code execution on hardware.

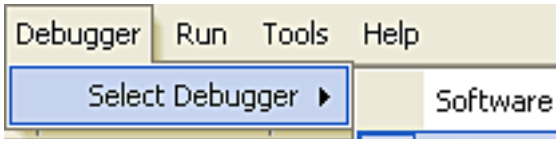
Step No. 1

If you have appropriate hardware and software for using the mikroICD select **mikroICD Debug** Build Type before compiling the project.



Step No. 2

Choosing the mikroICD Debug build type will select the mikroICD Debugger automatically. The Debugger Tool can also be selected from the **Debugger › Select Debugger** drop-down menu.



Compile the project and program dsPIC30/33 or PIC24 MCU.

Step No. 3


Start Debugger

Run the mikroICD by selecting **Run › Start Debugger** from the drop-down menu or by clicking the Start Debugger Icon . Starting Debugger makes more options available: Step Into, Step Over, Run to Cursor, etc. Line that is to be executed is color highlighted (blue by default). There is also notification about the program execution, Watch Window Status Bar will be colored yellow during the program execution.

mikroICD Debugger Options

Name	Description	Function Key
`Start Debugger`	Start Debugger.	[F9]
`Run/Pause Debugger`	Run or pause Debugger.	[F6]
`Stop Debugger`	Stop Debugger.	[Ctrl+F2]
`Toggle Breakpoints`	Toggle breakpoint at the current cursor position. To view all breakpoints, select Run › View Breakpoints from the drop-down menu. Double clicking an item in the Breakpoints Window List locates the breakpoint.	[F5]
`Run to cursor`	Execute all instructions between the current instruction and cursor position.	[F4]
`Step Into`	Execute the current Basic (single or multi–cycle) instruction, then halt. If the instruction is a routine call, enter the routine and halt at the first instruction following the call.	[F7]
`Step Over`	Execute the current Basic (single or multi–cycle) instruction, then halt.	[F8]
`Step Out`	Execute all remaining instructions in the current routine, return and then halt.	[Ctrl+F8]

Note: when the mikroICD is halted MCU peripherals are not active.

mikroICD Debugger Example

Here is a step-by-step mikroICD Debugger Example.

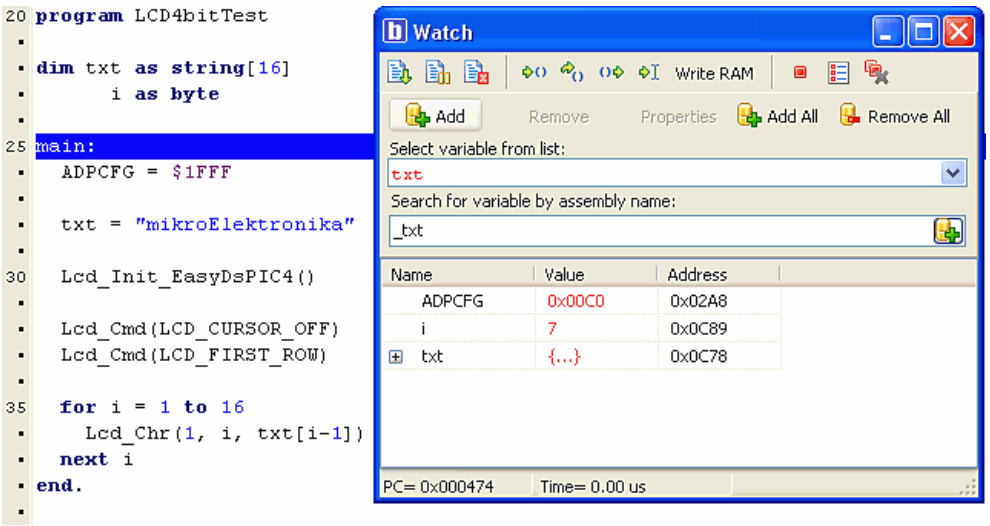
Step No. 1

First you have to write a program. By using this example we will show how the mikroICD operates:

```
program LCD4bitTest
dim txt as string[ 16]
       i as byte
main:
  ADPCFG = $1FFF
  txt = "mikroElektronika"
  Lcd_Init_EasyDsPIC4()
  Lcd_Cmd(LCD_CURSOR_OFF)
  Lcd_Cmd(LCD_FIRST_ROW)
  for i = 1 to 16
    Lcd_Chr(1, i, txt[ i-1] )
  next i
end.
```

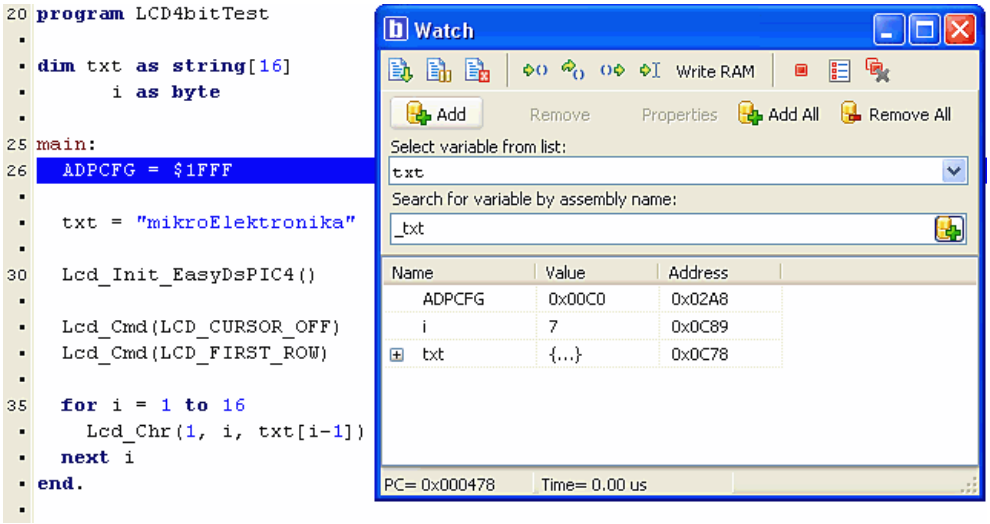
Step No. 2

After successful compilation and MCU programming press **F9** to start the mikroICD. After the mikroICD initialization a blue active line should appear.

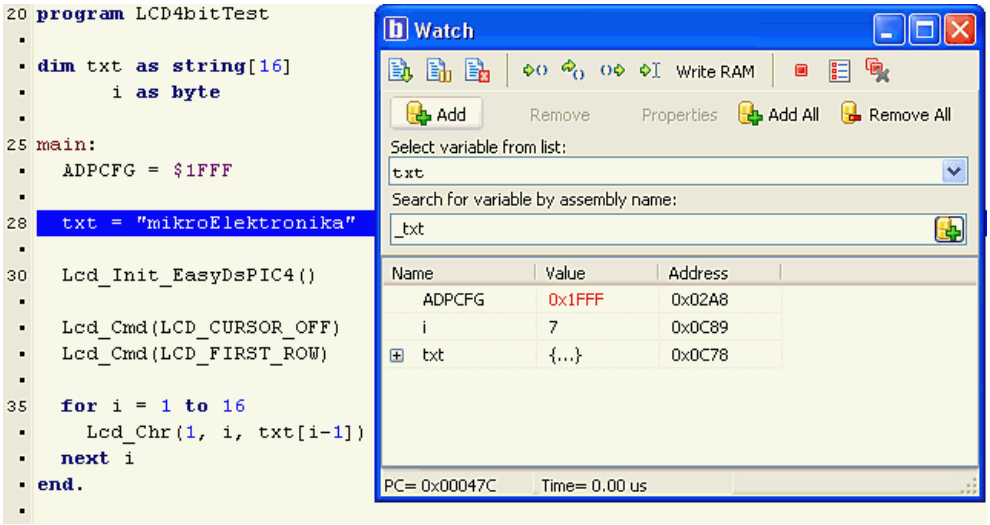


Step No. 3

We will debug the program line by line. To execute code line by line press **[F8]**. It is not recommended to the user to use Step Into **[F7]** and Step Over **[F8]** over Delay routines and routines containing delays. In this case the user should use Run to cursor **[F4]** function or Run **[F6]** function combined with Breakpoints.

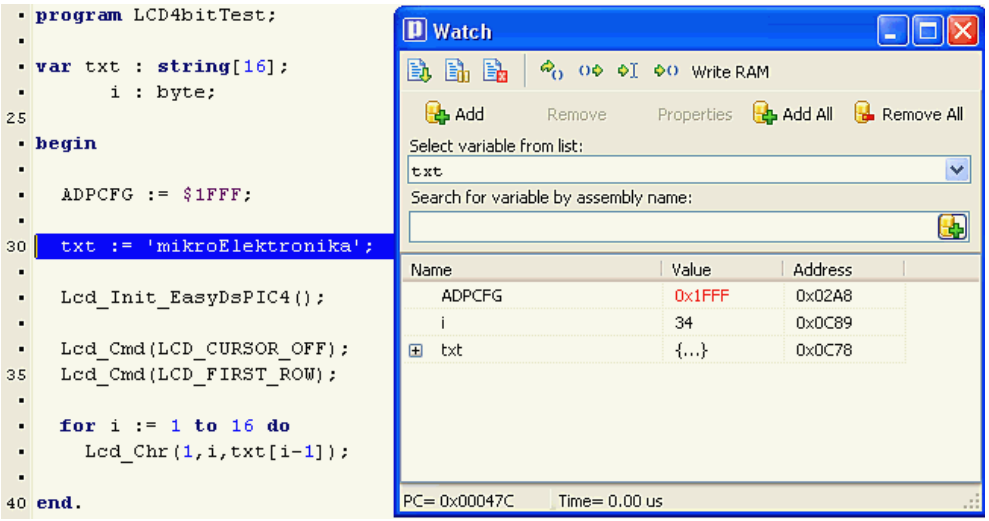


All changes are read from MCU and loaded into the Watch Window. Note that **ADPCFG** has changed its value to 0x1FFF.



Step No. 4

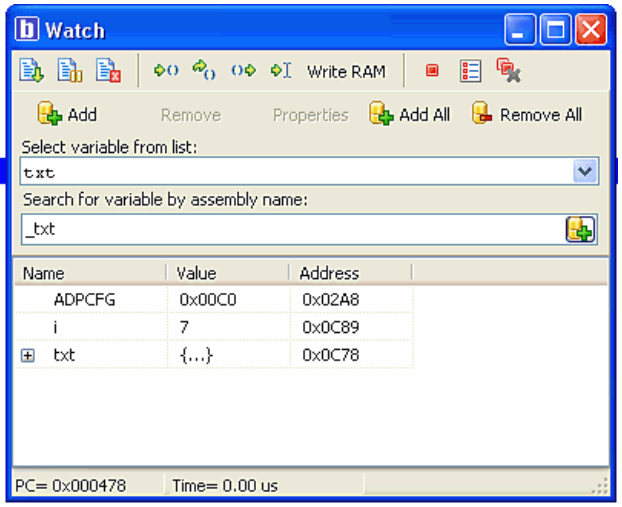
Step Into [**F7**], Step Over [**F8**] and Step Out [**Ctrl+F8**] are mikroICD debugger functions that are used in stepping mode. There is also a Real-Time mode supported by the mikroICD. Functions that are used in the Real-Time mode are Run/Pause Debugger [**F6**] and Run to cursor [**F4**]. Pressing **F4** executes the code until the program reaches the cursor position line.



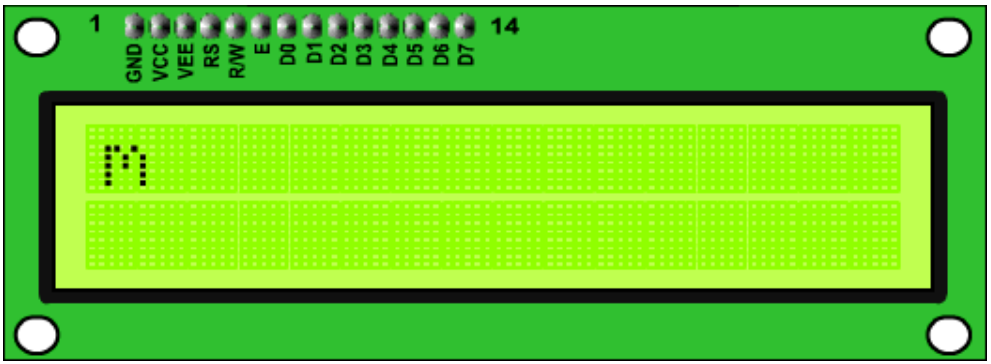
Step No. 5

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

```
20 program LCD4bitTest
   dim txt as string[16]
         i as byte
25 main:
     ADPCFG = $1FFF
     txt = "mikroElektronika"
30   Lcd_Init_EasyDsPIC4()
32   Lcd_Cmd(LCD_CURSOR_OFF)
     Lcd_Cmd(LCD_FIRST_ROW)
35   for i = 1 to 16
       Lcd_Chr(1, i, txt[i-1])
     next i
   end.
```



Breakpoints are divided into two groups: hardware and software breakpoints. The hardware breakpoints are placed in the MCU and provide fastest debugging. Number of hardware breakpoints is limited. There are 4 for the PIC24 and dsPIC33 family while for the dsPIC30 family this number depends on the MCU used. If all hardware breakpoints are used, then the next breakpoint will be software breakpoint. These breakpoints are placed inside the mikroICD and simulate hardware breakpoints. Software breakpoints are much slower than hardware breakpoints. These differences between hardware and software breakpoints are not visible in the mikroICD software but their different timings are quite notable.That's why it is important to know that there are two types of breakpoints.



mikroICD (In-Circuit Debugger) Overview

Watch Window

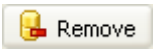
Debugger Watch Window is the main Debugger window which allows you to monitor program execution. To show the Watch Window, select **Debug Windows › Watch Window** from the **View** drop-down menu.

The Watch Window displays variables and registers of the MCU, along with their addresses and values. Values are updated along with the mikroICD code execution on hardware. Recently changed items are coloured red.

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

ADD Button

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



Remove Button

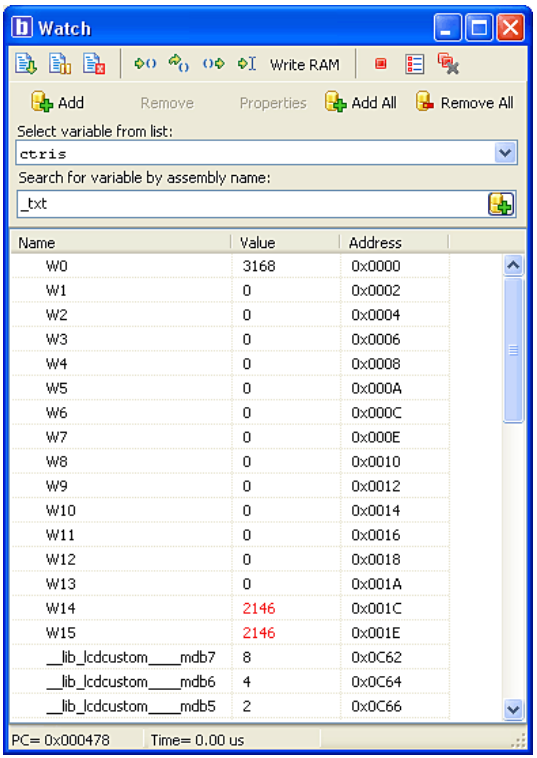
Variables can also be removed from the Watch Window. You just need to select the variable that you want to remove and click the Remove Button .

Add All Button

Add All Button adds all variables.

Remove All Button

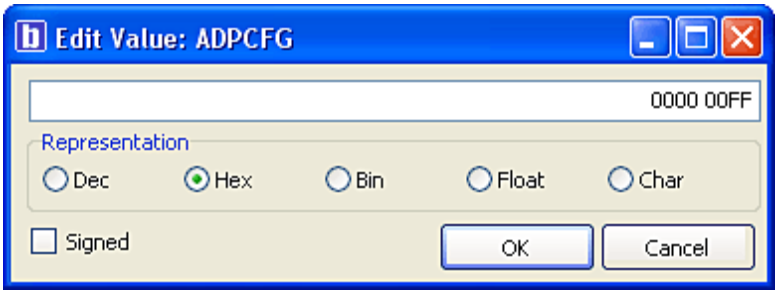
Remove All Button removes all variables.



Properties Button

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

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

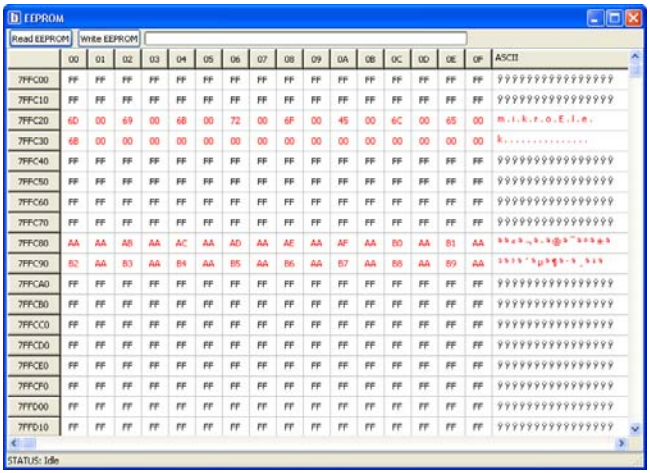


EEPROM Window

To show the mikroICD EEPROM Window, select **Debug Windows › EEPROM** from the **View** drop-down menu.

The EEPROM window shows current content of the MCU's internal EEPROM. There are two action buttons concerning the EEPROM watch window :

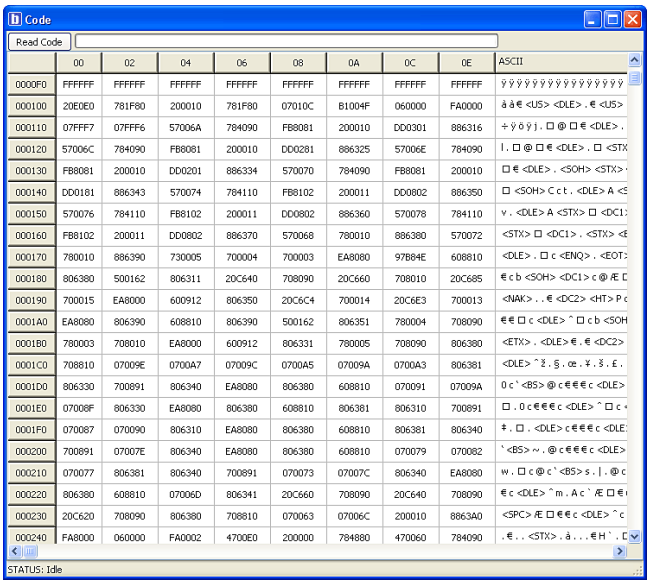
- **Flush EEPROM** writes data from the EEPROM window into MCU's internal EEPROM.
- **Read EEPROM** reads data from MCU's internal EEPROM and loads it up into the EEPROM window.



Code Window

To show the mikroICD Code Window, select **Debug Windows › Code** from the **View** drop-down menu. The Code window shows code (hex format) written into the MCU. There is an action button concerning the Code window :

Read Code reads code from the MCU and loads it up into the Code Window. Code reading is resources-consuming operation, so the user should wait until the reading is over.

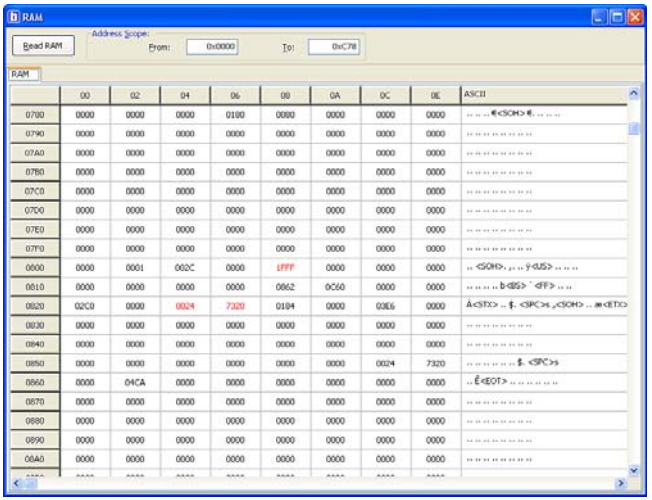


RAM Window

To show the mikroICD RAM window, select **Debug Windows › RAM** from the **View** drop-down menu.

The RAM Window displays the map of MCU's RAM, with recently changed items colored red.

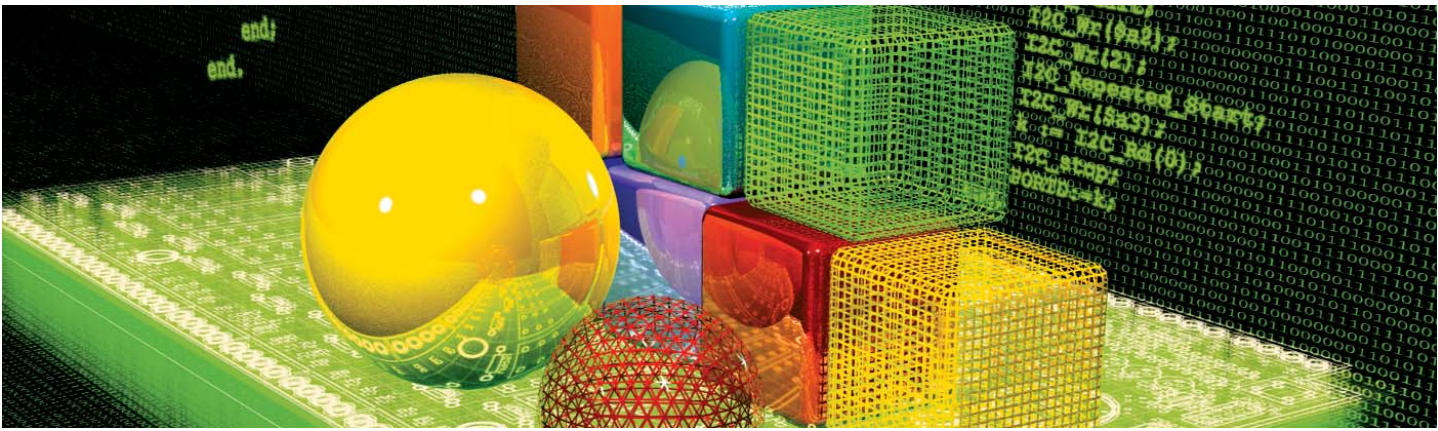
The user can edit and change the values in the RAM window whose content will be written to the MCU before the next instruction execution.



Common Errors

- Programming the MCU while the mikroICD is active.
- Debugging Release build version of the program with the mikroICD debugger.
- Debugging program code which has been changed, but has not been compiled and programmed into the MCU.
- Selecting an empty line in the code for Run to cursor [**F4**] and Toggle Breakpoints [**F5**] functions.
- Trying to Step Into [**F7**] the mikroBasic for dsPIC30/33 and PIC24 Library routines. Use Step Over [**F8**] command for these routines

CHAPTER 3



mikroBasic Language Reference

Why BASIC in the first place? The answer is simple: it is legible, easy-to-learn, structured programming language, with sufficient power and flexibility needed for programming microcontrollers. Whether you had any previous programming experience, you will find that writing programs in mikroBasic is very easy. This chapter will help you learn or recollect BASIC syntax, along with the specifics of programming dsPIC30/33 and PIC24 microcontrollers.

Predefined Globals and Constants

In order to facilitate dsPIC30/33 and PIC24 programming, mikroBasic for dsPIC30/33 and PIC24 implements a number of predefined globals and constants.

SFRs and related constants

All dsPIC30/33 and PIC24 SFRs are implicitly declared as global variables of volatile word type. These identifiers have an external linkage, and are visible in the entire project. When creating a project, the mikroBasic for dsPIC30/33 and PIC24 will include an appropriate (*.dbas) file from defs folder, containing declarations of available SFRs and constants (such as PORTB, ADPCFG, etc). All identifiers are in upper case, identical to nomenclature in the Microchip datasheets.

For a complete set of predefined globals and constants, look for "Defs" in the mikroBasic for dsPIC30/33 and PIC24 installation folder, or probe the Code Assistant for specific letters (Ctrl+Space in the Code Editor).

Math constants

In addition, several commonly used math constants are predefined in mikroBasic for dsPIC30/33 and PIC24:

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

Accessing Individual Bits

mikroBasic for dsPIC30/33 and PIC24 allows you to access individual bits of 16-bit variables. Simply use the direct member selector (.) with a variable, followed by a number `0..15`, with 15 being the most significant bit.

For example:

```
' If RB0 is set, set RC0:
if PORTB.0 = 1 then
  PORTC.0 = 1
end if
```

There is no need of any special declarations. This kind of selective access is an intrinsic feature of mikroBasic for dsPIC30/33 and PIC24 and can be used anywhere in the code.

Constants with value known in the compile-time can be used instead of numbers `0..15`, for example:

```
const pin5 = 5
...
' Set PORTB pin5
PORTB.pin5 = 1
```

Interrupts

The dsPIC30/33 and PIC24 interrupt controller module reduces numerous peripheral interrupt request signals to a single interrupt request signal to the dsPIC30/33 and PIC24 CPU and has the following features:

- Up to 8 processor exceptions and software traps
- 7 user-selectable priority levels
- Interrupt Vector Table (IVT) with up to 62 vectors (dsPIC30) or up to 118 vectors (dsPIC33 and PIC24)
- A unique vector for each interrupt or exception source
- Fixed priority within a specified user priority level
- Alternate Interrupt Vector Table (AIVT) for debug support

ISRs are organized in IVT. ISR is defined as a standard function but with the org directive following it, connecting the function with specific interrupt vector. For example `org 0x1A` is IVT address of Timer1 interrupt source of the dsPIC 30F3014 MCU. For more information on IVT refer to the dsPIC30/33 and PIC24 Family Reference Manual.

Function Calls from Interrupt

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

Example

Here is a simple procedure which inverts state of PORTB upon each interrupt from Timer1:

```
program TestTimer1

sub procedure Timer1Int org $1A
  '** it is necessary to clear manually the interrupt flag:
  IFS0 = IFS0 and $FFF7    ' Clear TMR1IF
  '** user code starts here
  LATB = not PORTB         ' Invert PORTB
  '** user code ends here
end sub
```

Code Optimization

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

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

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

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

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

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

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

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

Better code generation and local optimization
Code generation is more consistent and more attention is payed to implement specific solutions for the code "building bricks" that further reduce output code size.

Creating User's Libraries

mikroBasic for dsPIC30/33 and PIC24 allows you to create your own libraries. In order to create a library in mikroBasic for dsPIC30/33 and PIC24 follow the steps bellow:

- Create a new Basic file, see Managing Source Files
- Save the file in the compiler's Uses folder:
  DriveName:\Program Files\Mikroelektronika\mikroBasic dsPIC\Uses\
- Write a code for your library and save it
- Open the definition file for the MCU that you want to use.
  This file is placed in the compiler's Defs folder:
  DriveName:\Program Files\Mikroelektronika\mikroBasic dsPIC\Defs\
  and it is named p*MCU_NAME*def.dbas, for example p30F4013def.dbas
- Add the Name_Of_Your_Library to the definition file for the MCU:
  Repeat this step for each MCU that you want to use with your library
- Restart the compiler

dsPIC30/33 and PIC24 Specifics

In order to get the most from the mikroBasic for dsPIC30/33 and PIC24 compiler, the user should be familiar with certain aspects of dsPIC30/33 and PIC24 MCU. This knowledge is not essential, but it can provide a better understanding of the dsPIC30/33 and PIC24's capabilities and limitations, and their impact on the code writing as well.

Types Efficiency

First of all, the user should know that dsPIC30/33 and PIC24's ALU, which performs arithmetic operations, is optimized for working with 16-bit types. Although mikroBasic for dsPIC30/33 and PIC24 is capable of handling types like byte, char or short, dsPIC30/33 and PIC24 will generate a better code for 16-bit types word and integer type so use byte, char and short only in places where you can significantlly save RAM (e.g. for arrays dim a as byte[30]).

Nested Calls Limitations

There are no Nested Calls Limitations, except by RAM size. A Nested call represents a function call within the function body, either to itself (recursive calls) or to another function.

Recursive calls, as a form of cross-calling, are supported by mikroBasic for dsPIC30/33 and PIC24, but they should be used very carefully due to dsPIC30/33 and PIC24 stack and memory limitations. Also calling functions from interrupt is allowed. Calling function from both interrupt and main thread is allowed. Be carefull because this programming technique may cause unpredictable results if common resources are used in both main and interrupt.

Limits of Indirect Approach Through PSV

Constant aggregates are stored in Flash and are accessible trough PSV, which means that there can be max 32KByte of constants.

LEXICAL ELEMENTS

These topics provide a formal definition of the mikroBasic for dsPIC30/33 and PIC24 lexical elements. They describe different categories of word-like units (tokens) recognized by the language.

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

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

Whitespace

Whitespace is a collective name given to spaces (blanks), horizontal and vertical tabs, and comments. Whitespace serves to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded.

For example, the two sequences

```
dim tmp as byte
dim j as word
```

and

```
dim  tmp  as  byte
dim   j  as  word
```

are lexically equivalent and parse identically.

Newline Character

Newline character (CR/LF) is not a whitespace in BASIC, and serves as a statement terminator/separator. In mikroBasic for dsPIC30/33 and PIC24, however, you may use newline to break long statements into several lines. Parser will first try to get the longest possible expression (across lines if necessary), and then check for statement terminators.

Whitespace in Strings

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

```
some_string = "mikro foo"
```

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

```
some_string
=
"mikro foo"
newline character
```

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.

Use the apostrophe to create a comment:

```
' Any text between an apostrophe and the end of the
' line constitutes a comment. May span one line only.
```

There are no multi-line comments in mikroBasic for dsPIC30/33 and PIC24.

TOKENS

Token is the smallest element of a mikroBasic for dsPIC30/33 and PIC24 program, 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.

mikroBasic for dsPIC30/33 and PIC24 recognizes the following kinds of tokens:

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

Token Extraction Example

Here is an example of token extraction. See the following code sequence:

```
end_flag = 0
```

The compiler would parse it into four tokens:

```
end_flag      ' variable identifier
=             ' assignment operator
0             ' literal
newline       ' statement terminator
```

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

LITERALS

Literals are tokens representing fixed numeric or character values.

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

Integer Literals

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

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

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

The percent-sign prefix (%) indicates a binary numeral (for example, %0101).

Here are some examples:

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

The allowed range of values is imposed by the largest data type in mikroBasic for dsPIC30/33 and PIC24 – `longword`. The compiler will report an error if the literal exceeds `4294967295` (`$FFFFFFFF`).

Floating Point Literals

A floating-point value consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- e or E and a signed integer exponent (optional)

You can omit either decimal integer or decimal fraction (but not both).

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

mikroBasic limits floating-point constants to the range of
±1.17549435082 * 10^-38 .. ±6.80564774407 * 10^38.

Here are some examples:

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

Character Literals

Character literal is one character from the extended ASCII character set, enclosed with quotes (for example, "A"). Character literal can be assigned to variables of byte and char type (variable of byte will be assigned the ASCII value of the character). Also, you can assign character literal to a string variable.

String Literals

String literal is a sequence of characters from the extended ASCII character set, enclosed with quotes. Whitespace is preserved in string literals, i.e. parser does not "go into" strings but treats them as single tokens.

Length of string literal is a number of characters it consists of. String is stored internally as the given sequence of characters plus a final null character. This null character is introduced to terminate the string, it does not count against the string's total length.

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

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

Here are several string literals:

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

The quote itself cannot be a part of the string literal, i.e. there is no escape sequence. You could use the built-in function Chr to print a quote: Chr(34). Also, see String Splicing.

KEYWORDS

Keywords are special-purpose words which cannot be used as normal identifier names.

Beside standard BASIC keywords, all relevant SFR are defined as global variables and represent reserved words that cannot be redefined (for example: `PORTB`, `TMR1`, `T1CON`, etc). Probe Code Assistant for specific letters (Ctrl+Space in Editor) or refer to Predefined Globals and Constants.

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

Here is the alphabetical listing of keywords in mikroBasic for dsPIC30/33 and PIC24:

abs	datedf	include	nothing	short
and	datepart	input	now	sin
appactivate	dateserial	instr	nper	single
array	datevalue	int	npv	sln
as	ddb	integer	object	space
asc	deftype	ipmt	oct	spc
asm	dim	irr	on	sqr
atn	dir	is	open	static
attribute	div	isarray	option	step
base	do	isdate	or	stop
beep	doevents	isempty	org	str
boolean	double	iserror	pmt	strcomp
byref	each	ismissing	ppmt	strconv
byte	else	isnull	print	string
call	empty	isnumeric	private	sub
case	end	isobject	procedure	switch
cbool	environ	kill	property	syd
cbyte	eof	lbound	pset	system
ccur	eqv	lcase	public	tab
cdate	erase	left	put	tan
cdate	err	len	pv	then
cdbl	error	let	qbcolor	time
char	exit	line	raise	timer
chdir	exp	loc	randomize	timeserial
chdrive	explicit	lock	rate	timevalue
chr	fileattr	lof	real	to
cint	filecopy	log	redim	trim
circle	filedatetime	long	register	typedef
class	filelen	longint	rem	typename
clear	fix	loop	reset	ubound
clng	float	lset	resume	ucase
close	for	ltrim	return	unlock
command	form	me	rgb	until
compare	format	mid	right	val
const	freefile	minute	rmdir	variant
cos	fv	mirr	rnd	vartype
createobject	get	mkdir	rset	version
csng	getattr	mod	rtrim	volatile
cstr	getobject	module	second	weekday
curdir	gosub	month	seek	wend
currency	goto	msgbox	select	while
cvar	hex	name	sendkeys	width
cverr	hour	new	set	with
date	if	next	setattr	word
dateadd	iif	not	sgn	write
	imp	not	shell	xor

IDENTIFIERS

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

Identifiers can contain letters from a to z and A to Z, the underscore character "_" and digits from 0 to 9. First character must be a letter or an underscore, i.e. identifier cannot begin with a numeral.

Case Sensitivity

mikroBasic for dsPIC30/33 and PIC24 is not case sensitive, so `Sum`, `sum`, and `suM` are equivalent identifiers.

Uniqueness and Scope

Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same scope. Simply, duplicate names are *illegal* within the same scope. For more information, refer to Scope and Visibility.

Identifier Examples

Here are some valid identifiers:

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

… and here are some invalid identifiers:

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

PUNCTUATORS

The mikroBasic punctuators (also known as separators) are:

`[ ]` – Brackets
`()` – Parentheses
`,` – Comma
`:` – Colon
`.` – Dot

Brackets

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

```
dim alphabet as byte[ 30]
' ...
alphabet[ 2]  = "c"
```

For more information, refer to Arrays.

Parentheses

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

```
d = c *  (a + b)                ' Override normal precedence
if (d = z) then ...             ' Useful with conditional statements
func()                          ' Function call, no arguments
sub function func2(dim n as word)
                                ' Function declaration w/ parameters
```

For more information, refer to Operators Precedence and Associativity, Expressions, or Functions and Procedures.

Comma

Comma (`,`) separates the arguments in function calls:

```
Lcd_Out(1, 1, txt)
```

Furthermore, the comma separates identifiers in declarations:

```
dim i, j, k as word
```

The comma also separates elements in initialization lists of constant arrays:

```
const MONTHS as byte[ 12] = (31,28,31,30,31,30,31,31,30,31,30,31)
```

Colon

Colon (:) is used to indicate a labeled statement:

```
start:  nop
  '...
goto start
```

For more information, refer to Labels.

Dot

Dot (.) indicates access to a structure member. For example:

```
person.surname = "Smith"
```

For more information, refer to Structures.

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

PROGRAM ORGANIZATION

mikroBasic for dsPIC30/33 and PIC24 imposes strict program organization. Below you can find models for writing legible and organized source files. For more information on file inclusion and scope, refer to Modules and to Scope and Visibility.

Organization of Main Module

Basically, a main source file has two sections: declaration and program body. Declarations should be in their proper place in the code, organized in an orderly manner. Otherwise, the compiler may not be able to comprehend the program correctly.

When writing code, follow the model presented below.

Main module should look like this:

```
program <program name>
include <include other modules>

'*******************************************************
'* Declarations (globals):
'*******************************************************

' symbols declarations
symbol ...

' constants declarations
const ...

' structures declarations
structure ...

' variables declarations
dim ...

' procedures declarations
sub procedure procedure_name(...)
  <local declarations>
  ...
end sub

' functions declarations
sub function function_name(...) as return_type
  <local declarations>
  ...
end sub

'*******************************************************
'* Program body:
'*******************************************************

main:
  ' write your code here
end.
```

Organization of Other Modules

Modules other than main start with the keyword `module`. Implementation section starts with the keyword `implements`. Follow the model presented below.

Other modules should look like this:

```
module  <module name>
include <include other modules>

'*******************************************************
'* Interface (globals):
'*******************************************************
' symbols declarations
symbol ...

' constants declarations
const ...

' structures declarations
structure ...

' variables declarations
dim ...

' procedures prototypes
sub procedure procedure_name(...)

' functions prototypes
sub function function_name(...) as return_type

'*******************************************************
'* Implementation:
'*******************************************************

implements

' constants declarations
const ...

' variables declarations
dim ...

' procedures declarations
sub procedure procedure_name(...)
  <local declarations>
  ...
end sub

' functions declarations
sub function function_name(...) as return_type
  <local declarations>
  ...
end sub
end.
```

SCOPE AND VISIBILITY

Scope

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

Place of declaration	Scope
Identifier is declared in the declaration section of the main module, out of any function or procedure.	Scope extends from the point where it is declared to the end of the current file, including all routines enclosed within that scope. These identifiers have a *file scope* and are referred to as *globals*.
Identifier is declared in the function or procedure	Scope extends from the point where it is declared to the end of the current routine. These identifiers are referred to as *locals*.
Identifier is declared in the interface section of the module	Scope extends the interface section of a module from the point where it is declared to the end of the module, and to any other module or program that uses that module. The only exception are symbols which have a scope limited to the file in which they are declared.
Identifier is declared in the implementation section of the module, but not within any function or procedure	Scope extends from the point where it is declared to the end of the module. The identifier is available to any function or procedure in the module.

Visibility

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

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the appearance of a duplicate identifier: 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.

In mikroBasic for dsPIC30/33 and PIC24, each project consists of a single project file and one or more module files. The project file, with extension `.dbp` contains information on the project, while modules, with extension `.dbas`, contain the actual source code. See Program Organization for a detailed look at module arrangement.

Modules allow you to:

- break large programs into encapsulated modules that can be edited separately,
- create libraries that can be used in different projects,
- distribute libraries to other developers without disclosing the source code.

Each module is stored in its own file and compiled separately; compiled modules are linked to create an application. To build a project, the compiler needs either a source file or a compiled module file for each module.

Include Clause

mikroBasic includes modules by means of the include clause. It consists of the reserved word include, followed by a quoted module name. Extension of the file should not be included.

You can include one file per include clause. There can be any number of the include clauses in each source file, but they all must be stated immediately after the program (or module) name.

Here's an example:

```
program MyProgram

include "utils"
include "strings"
include "MyUnit"

...
```

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

- If both `.dbas` and `.mcl` files are found, the compiler will check their dates and include the newer one in the project. If the `.dbas` file is newer than the `.mcl`, then `.dbas` file will be recompiled and new `.mcl` will be created, overwriting the old `.mcl`.
- If only the `.dbas` file is found, the compiler will create the `.mcl` file and include it in the project;
- If only the `.mcl` file is present, i.e. no source code is available, the compiler will include it as found;
- If none of the files found, the compiler will issue a "File not found" warning.

Main Module

Every project in mikroBasic requires a single main module file. The main module is identified by the keyword program at the beginning. It instructs the compiler where to "start".

After you have successfully created an empty project with Project Wizard, Code Editor will display a new main module. It contains the bare-bones of the program:

```
program MyProject

   ' main procedure
main:
   ' Place program code here
end.
```

Other than comments, nothing should precede the keyword program. After the program name, you can optionally place the `include` clauses.

Place all global declarations (constants, variables, labels, routines, structures) before the label `main`.

Other Modules

Modules other than main start with the keyword `module`. Newly created blank module contains the bare-bones:

```
module MyModule

implements

end.
```

Other than comments, nothing should precede the keyword `module`. After the module name, you can optionally place the `include` clauses.

Interface Section

Part of the module above the keyword `implements` is referred to as *interface* section. Here, you can place global declarations (constants, variables, labels, routines, structures) for the project.

Do not define routines in the interface section. Instead, state the prototypes of routines (from implementation section) that you want to be visible outside the module. Prototypes must exactly match the declarations.

Implementation Section

Implementation section hides all the irrelevant innards from other modules, allowing encapsulation of code.

Everything declared below the keyword `implements` is *private*, i.e. has its scope limited to the file. When you declare an identifier in the implementation section of a module, you cannot use it outside the module, but you can use it in any block or routine defined within the module.

By placing the prototype in the interface section of the module (above the `implements`) you can make the routine *public*, i.e. visible outside of module. Prototypes must exactly match the declarations.

VARIABLES

Variable is an object whose value can be changed during the runtime. Every variable is declared under unique name which must be a valid identifier. This name is used for accessing the memory location occupied by the variable.

Variables are declared in the declaration part of the file or routine — each variable needs to be declared before it is used. Global variables (those that do not belong to any enclosing block) are declared below the include statements, above the label `main`.

Specifying a data type for each variable is mandatory. mikroBasic syntax for variable declaration is:

```
dim identifier_list as type
```

Here, *`identifier_list`* is a comma-delimited list of valid identifiers, and type can be any data type.

For more details refer to Types and Types Conversions. For more information on variables' scope refer to the chapter Scope and Visibility.

Here are a few examples:

```
dim i, j, k as byte
dim counter, temp as word
dim samples as longint[ 100]
```

Variables and dsPIC30/33 and PIC24

Every declared variable consumes part of RAM memory. Data type of variable determines not only the allowed range of values, but also the space a variable occupies in RAM memory. Bear in mind that operations using different types of variables take different time to be completed. mikroBasic for dsPIC30/33 and PIC24 recycles local variable memory space – local variables declared in different functions and procedures share the same memory space, if possible.

There is no need to declare SFR explicitly, as mikroBasic for dsPIC30/33 and PIC24 automatically declares relevant registers as global variables of `word`. For example: `W0, TMR1`, etc.

CONSTANTS

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

Constants are declared in the declaration part of the program or routine, with the following syntax:

```
const constant_name [ as type] = value
```

Every constant is declared under unique *constant_name* which must be a valid identifier. It is a tradition to write constant names in uppercase. Constant requires you to specify *value*, which is a literal appropriate for the given *type*. type is optional and in the absence of it , the compiler assumes the "smallest" type that can accommodate *value*.

Note: You cannot omit *type* if declaring a constant array.

Here are a few examples:

```
const MAX as longint = 10000
const MIN = 1000            ' compiler will assume word type
const SWITCH = "n"          ' compiler will assume char type
const MSG = "Hello"         ' compiler will assume string type
const MONTHS as byte[ 12] = (31,28,31,30,31,30,31,31,30,31,30,31)
```

LABELS

Labels serve as targets for the goto and gosub statements. Mark the desired statement with label and colon like this:

```
label_identifier : statement
```

No special declaration of label is necessary in mikroBasic for dsPIC30/33 and PIC24.

Name of the label needs to be a valid identifier. The labeled statement and `goto/gosub` statement must belong to the same block. Hence it is not possible to jump into or out of routine. Do not mark more than one statement in a block with the same label.

Note: The label `main` marks the entry point of a program and must be present in the main module of every project. See Program Organization for more information.

Here is an example of an infinite loop that calls the procedure `Beep` repeatedly:

```
loop:
  Beep
goto loop
```

SYMBOLS

mikroBasic symbols allow you to create simple macros without parameters. You can replace any line of code with a single identifier alias. Symbols, when properly used, can increase code legibility and reusability.

Symbols need to be declared at the very beginning of the module, right after the module name and (optional) include clauses. Check Program Organization for more details. Scope of a symbol is always limited to the file in which it has been declared.

Symbol is declared as:

```
symbol alias = code
```

Here, *alias* must be a valid identifier which you will use throughout the code. This identifier has a file scope. The *code* can be any line of code (literals, assignments, function calls, etc).

Using a symbol in the program consumes no RAM – the compiler will simply replace each instance of a symbol with the appropriate line of code from the declaration.

Here is an example:

```
symbol MAXALLOWED = 216            ' Symbol as alias for numeric
                                   ' value
symbol PORT = PORTC                ' Symbol as alias for SFR
symbol MYDELAY = Delay_ms(1000)    ' Symbol as alias for procedure
                                   ' call

dim cnt as byte   ' Some variable

'...
main:

if cnt > MAXALLOWED then
  cnt = 0
  PORT.1 = 0
  MYDELAY
end if
```

Note: Symbols do not support macro expansion in a way the C preprocessor does.

FUNCTIONS AND PROCEDURES

Functions and procedures, collectively referred to as *routines*, are subprograms (self-contained statement blocks) which perform a certain task based on a number of input parameters. When executed, a function returns value while procedure does not.

Functions

Function is declared like this:

```
sub function function_name(parameter_list) as return_type
  [  local declarations ]
   function body
end sub
```

`function_name` represents a function's name and can be any valid identifier. `return_type` is a type of return value and can be any simple type. Within parentheses, `parameter_list` is a formal parameter list similar to variable declaration. In mikroBasic for dsPIC30/33 and PIC24, parameters are always passed to a function by value. To pass an argument by address, add the keyword byref ahead of identifier.
`Local declarations` are optional declarations of variables and/or constants, local for the given function. `Function body` is a sequence of statements to be executed upon calling the function.

Calling a function

A function is called by its name, with actual arguments placed in the same sequence as their matching formal parameters. The compiler is able to coerce mismatching arguments to the proper type according to implicit conversion rules. Upon a function call, all formal parameters are created as local objects initialized by values of actual arguments. Upon return from a function, a temporary object is created in the place of the call and it is initialized by the value of the function result. This means that function call as an operand in complex expression is treated as the function result. In standard Basic, a `function_name` is automatically created local variable that can be used for returning a value of a function. mikroBasic for dsPIC30/33 and PIC24 also allows you to use the automatically created local variable result to assign the return value of a function if you find function name to be too ponderous. If the return value of a function is not defined the compiler will report an error.

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

Example

Here's a simple function which calculates xn based on input parameters x and n (n > 0):

```
sub function power(dim x, n as byte) as longint
dim i as byte
  result = 1
  if n > 0 then
    for i = 1 to n
      result = result*x
    next i
  end if
end sub
```

Now we could call it to calculate, say, 3^{12}:

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

Procedures

Procedure is declared like this:

```
sub procedure procedure_name(parameter_list)
  [ local declarations ]
  procedure body
end sub
```

procedure_name represents a procedure's name and can be any valid identifier. Within parentheses, *parameter_list* is a formal parameter list similar to variable declaration. In mikroBasic for dsPIC30/33 and PIC24, parameters are always passed to procedure by value; to pass argument by address, add the keyword byref ahead of identifier.

Local declarations are optional declaration of variables and/or constants, local for the given procedure. *Procedure body* is a sequence of statements to be executed upon calling the procedure.

Calling a procedure

A procedure is called by its name, with actual arguments placed in the same sequence as their matching formal parameters. The compiler is able to coerce mismatching arguments to the proper type according to implicit conversion rules. Upon procedure call, all formal parameters are created as local objects initialized by values of actual arguments.

Procedure call is a self-contained statement.

Example

Here's an example procedure which transforms its input time parameters, preparing them for output on LCD:

```
sub procedure time_prep(dim byref sec, min, hr as byte)
  sec  = ((sec and $F0) >> 4)*10 + (sec and $0F)
  min  = ((min and $F0) >> 4)*10 + (min and $0F)
  hr   = ((hr and $F0) >> 4)*10 + (hr  and $0F)
end sub
```

mikroBasic for dsPIC30/33 and PIC24 is strictly typed language, which means that every variable and constant need to have a strictly defined type, known at the time of compilation.

The type serves:

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

mikroBasic for dsPIC30/33 and PIC24 supports many standard (predefined) and user-defined data types, including signed and unsigned integers of various sizes, arrays, strings, pointers and structures.

Type Categories

Types can be divided into:

- simple types
- arrays
- strings
- pointers
- structures (user defined types)

SIMPLE TYPES

Simple types represent types that cannot be divided into more basic elements and are the model for representing elementary data on machine level. Basic memory unit in mikroBasic for dsPIC30/33 and PIC24 has 16 bits.

Here is an overview of simple types in mikroBasic for dsPIC30/33 and PIC24:

Type	Size	Range
`byte, char`	8-bit	0 .. 255
`short`	8-bit	-127 .. 128
`word`	16-bit	0 .. 65535
`integer`	16-bit	-32768 .. 32767
`longword, dword`	32-bit	0 .. 4294967295
`longint`	32-bit	-2147483648 .. 2147483647
`float`	32-bit	$\pm 1.17549435082 * 10^{-38}$ .. $\pm 6.80564774407 * 10^{38}$

You can assign signed to unsigned or vice versa only using the explicit conversion. Refer to Types Conversions for more information.

An array represents an indexed collection of elements of the same type (called the base type). Since each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once.

Array Declaration

Array types are denoted by constructions of the form:

```
type[ array_length]
```

Each of elements of an array is numbered from 0 through `array_length` - 1. Every element of an array is of `type` and can be accessed by specifying array name followed by element's index within brackets.

Here are a few examples of array declaration:

```
dim weekdays as byte[ 7]
dim samples  as word[ 50]

main:
  ' Now we can access elements of array variables, for example:
  samples[ 0]  = 1
  if samples[ 37]  = 0 then
    ' ...
```

Constant Arrays

Constant array is initialized by assigning it a comma-delimited sequence of values within parentheses. For example:

```
'Declare a constant array which holds number of days in each month:
const MONTHS as byte[ 12]  = (31,28,31,30,31,30,31,31,30,31,30,31)
```

Note that indexing is zero based; in the previous example, number of days in January is `MONTHS[ 0]` and number of days in December is `MONTHS[ 11]` .

The number of assigned values must not exceed the specified length. Vice versa is possible, when the trailing "excess" elements will be assigned zeroes.

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

STRINGS

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

```
string[ string_length]
```

The specifier *string_length* is a number of characters a string consists of. The string is stored internally as the given sequence of characters plus a final `null` character (zero). This appended "stamp" does not count against string's total length.

A null string (`""`) is stored as a single `null` character.

You can assign string literals or other strings to string variables. The string on the right side of an assignment operator has to be shorter than another one, or of equal length. For example:

```
dim msg1 as string[ 20]
dim msg2 as string[ 19]

main:
  msg1 = "This is some message"
  msg2 = "Yet another message"
  msg1 = msg2 ' this is ok, but vice versa would be illegal
```

Alternately, you can handle strings element–by–element. For example:

```
dim s as string[ 5]
' ...
s = "mik"
' s[0] is char literal "m"
' s[1] is char literal "i"
' s[2] is char literal "k"
' s[3] is zero
' s[4] is undefined
' s[5] is undefined
```

Be careful when handling strings in this way, since overwriting the end of a string will cause an unpredictable behavior.

Note
mikroBasic for dsPIC30/33 and PIC24 includes String Library which automatizes string related tasks.

POINTERS

A pointer is a data type which holds a memory address. While a variable accesses that memory address directly, a pointer can be thought of as a reference to that memory address.

To declare a pointer data type, add a carat prefix (^) before type. For example, if you are creating a pointer to an `integer`, you would write:

```
^integer
```

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

```
dim p as ^word
'...
p^ = 5
```

A pointer can be assigned to another pointer. However, note that only address, not value, is copied. Once you modify the data located at one pointer, the other pointer, when dereferenced, also yields modified data.

@ Operator

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

- If `X` is a variable, `@X` returns the address of `X`.
- If `F` is a routine (a function or procedure), `@F` returns `F`'s entry point (the result is of longint).

STRUCTURES

A structure represents a heterogeneous set of elements. Each element is called a *member*; the declaration of a structure type specifies a name and type for each member. The syntax of a structure type declaration is

```
structure structname
  dim member1 as type1
  '...
  dim membern as typen
end structure
```

where `structname` is a valid identifier, each type denotes a type, and each `member` is a valid identifier. The scope of a member identifier is limited to the structure in which it occurs, so you don't have to worry about naming conflicts between member identifiers and other variables.

For example, the following declaration creates a structure type called `Dot`:

```
structure Dot
  dim x as float
  dim y as float
end structure
```

Each `Dot` contains two members: `x` and `y` coordinates; memory is allocated when you instantiate the structure, like this:

```
dim m, n as Dot
```

This variable declaration creates two instances of `Dot`, called `m` and `n`.

A member can be of the previously defined structure type. For example:

```
' Structure defining a circle:
structure Circle
  dim radius as float
  dim center as Dot
end structure
```

Structure Member Access

You can access the members of a structure by means of dot (.) as a direct member selector. If we had declared the variables `circle1` and `circle2` of the previously defined type `Circle`:

```
dim circle1, circle2 as Circle
```

we could access their individual members like this:

```
circle1.radius = 3.7
circle1.center.x = 0
circle1.center.y = 0
```

You can also commit assignments between complex variables, if they are of the same type:

```
circle2 = circle1   ' This will copy values of all members
```

TYPES CONVERSIONS

Conversion of variable of one type to variable of another type is typecasting. mikroBasic for dsPIC30/33 and PIC24 supports both implicit and explicit conversions for built-in types.

Implicit Conversion

Compiler will provide an automatic implicit conversion in the following situations:

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

Promotion

When operands are of different types, implicit conversion promotes the less complex to the more complex type taking the following steps:

```
byte/char     ->     word
short         ->     integer
short         ->     longint
integer       ->     longint
integral      ->     float
```

Higher bytes of extended unsigned operand are filled with zeroes. Higher bytes of extended signed operand are filled with bit sign (if number is negative, fill higher bytes with one, otherwise with zeroes).
Higher bytes of extended unsigned operand are filled with zeroes. Higher bytes of extended signed operand are filled with bit sign (if number is negative, fill higher bytes with one, otherwise with zeroes). For example:

```
dim a as byte
dim b as word
'...
a = $FF
b = a   ' a is promoted to word, b becomes $00FF
```

Clipping

In assignments and statements that require an expression of particular type, destination will store the correct value only if it can properly represent the result of expression, i.e. if the result fits in destination range.

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

```
dim i as byte
dim j as word
'...
j = $FF0F
i = j   ' i becomes $0F, higher byte $FF is lost
```

Explicit Conversion

Explicit conversion can be executed at any point by inserting type keyword (`byte`, `word`, `short`, `integer`, `longint`, or `float`) ahead of the expression to be converted. The expression must be enclosed in parentheses. Explicit conversion can be performed only on the operand left of the assignment operator.

Special case is the conversion between signed and unsigned types. Explicit conversion between signed and unsigned data does not change binary representation of data — it merely allows copying of source to destination.

For example:

```
dim a as byte
dim b as short
'...
b = -1
a = byte(b)   ' a is 255, not 1

' This is because binary representation remains
' 11111111; it's just interpreted differently now
```

You cannot execute explicit conversion on the operand left of the assignment operator:

```
word(b) = a   ' Compiler will report an error
```

OPERATORS

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

There are four types of operators in mikroBasic for dsPIC30/33 and PIC24:

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

Operators Precedence and Associativity

There are 4 precedence categories in mikroBasic for dsPIC30/33 and PIC24. Operators in the same category have equal precedence with each other.

Each category has an associativity rule: left-to-right, or right-to-left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

Precedence	Operands	Operators	Associativity
4	1	`@ not + -`	right-to-left
3	2	`* / div mod and << >>`	left-to-right
2	2	`+ - or xor`	left-to-right
1	2	`= <> < > <= >=`	left-to-right

Arithmetic Operators

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

All arithmetic operators associate from left to right.

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

Division by Zero

If 0 (zero) is used explicitly as the second operand (i.e. `x div 0`), the compiler will report an error and will not generate code. But in case of implicit division by zero: `x div y`, where y is 0 (zero), dsPIC MCU will generate the *division* by zero trap).

Unary Arithmetic Operators

Operator - can be used as a prefix unary operator to change sign of a signed value. Unary prefix operator + can be used, but it doesn't affect data.

For example:

```
b = -a;
```

Relational Operators

Use relational operators to test equality or inequality of expressions. All relational operators return TRUE or FALSE. All relational operators associate from left to right.

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

Relational Operators in Expressions

The equal sign (=) can also be an assignment operator, depending on context. Precedence of arithmetic and relational operators was designated in such a way to allow complex expressions without parentheses to have expected meaning:

```
if aa + 5 >= bb - 1.0 / cc then
                  ' same as: if (aa + 5) >= (bb - (1.0 / cc)) then
  dd = My_Function()
end if
```

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 not which associates from right to left.

Bitwise Operators Overview

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

Logical Operations on Bit Level

and	0	1
0	0	0
1	0	1

or	0	1
0	0	1
1	1	1

xor	0	1
0	0	1
1	1	0

not	0	1
	1	0

The bitwise operators `and`, `or`, and `xor` perform logical operations on the appropriate pairs of bits of their operands. The operator not complements each bit of its operand. For example:

```
$1234 and $5678          ' equals $1230

' because ..

' $1234 : 0001 0010 0011 0100
' $5678 : 0101 0110 0111 1000
' ----------------------------
'   and : 0001 0010 0011 0000

' .. that is, $1230

' Similarly:

$1234 or  $5678          ' equals $567C
$1234 xor $5678          ' equals $444C
not $1234                ' equals $EDCB
```

Unsigned and Conversions

If number is converted from less complex to more complex data type, the upper bytes are filled with zeroes. If number is converted from more complex to less complex data type, the data is simply truncated (upper bytes are lost).

For example:

```
dim a as byte
dim b as word
' ...
  a = $AA
  b = $F0F0
  b = b and a
  ' a is extended with zeroes; b becomes $00A0
```

Signed and Conversions

If number is converted from less complex to more complex data type, the upper bytes are filled with ones if sign bit is 1 (number is negative); the upper bytes are filled with zeroes if sign bit is 0 (number is positive). If number is converted from more complex to less complex data type, the data is simply truncated (the upper bytes are lost).

For example:

```
dim a as byte
dim b as word
' ...
  a = -12
  b = $70FF
  b = b and a

  ' a is sign extended, upper byte is $FF;
  ' b becomes $70F4
```

Bitwise Shift Operators

The binary operators << and >> move the bits of the left operand by a number of positions specified by the right operand, to the left or right, respectively. Right operand has to be positive and less than 255.

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

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

Boolean Operators

Although mikroBasic for dsPIC30/33 and PIC24 does not support the boolean type, you have Boolean operators at your disposal for building complex conditional expressions. These operators conform to standard Boolean logic, and return either `TRUE` (all ones) or `FALSE` (zero):

Operator	Operation
`and`	logical AND
`or`	logical OR
`xor`	logical exclusive OR (XOR)
`not`	logical negation

Boolean operators associate from left to right. The negation operator not associates from right to left.

EXPRESSIONS

An expression is a sequence of operators, operands, and punctuators that returns a value.

The *primary expressions* include: literals, constants, variables and function calls. From them, using operators, more complex expressions can be created. Formally, expressions are defined recursively: subexpressions can be nested up to the limits of memory.

Expressions are evaluated according to certain conversion, grouping, associativity and precedence rules that depend on the operators used, presence of parentheses, and data types of the operands. The precedence and associativity of the operators are summarized in Operator Precedence and Associativity. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by mikroBasic for dsPIC30/33 and PIC24.

STATEMENTS

Statements define algorithmic actions within a program. Each statement needs to be terminated by a newline character (CR/LF).

The simplest statements include assignments, routine calls and jump statements. They can be combined to form loops, branches and other structured statements. In the absence of specific jump and selection statements, they are executed sequentially in order of appearance in the source code.

Refer to:

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

Assignment Statements

Assignment statements have the following form:

```
variable = expression
```

The statement evaluates `expression` and assigns its value to `variable`. All rules of implicit conversion are applied. `Variable` can be any declared variable or array element, and `expression` can be any expression.

Do not confuse the assignment with relational operator = which tests for equality. mikroBasic for dsPIC30/33 and PIC24 will interpret the meaning of the character = from the context.

Conditional Statements

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

- if
- select case

If Statement

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

```
if expression then
  statements
[ else
  other statements]
end if
```

When *expression* evaluates to true, *statements* execute. If *expression* is false, *other statements* execute. The *expression* must convert to a boolean type; otherwise, the condition is ill-formed. The else keyword with an alternate block of statements (*other statements*) is optional.

Nested if statements

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

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

The compiler treats the construction in this way:

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

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

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

Select Case Statement

Use the *select case* statement to pass control to a specific program branch, based on a certain condition. The *select case* statement consists of selector expression (condition) and list of possible values. The syntax of the *select case* statement is:

```
select case selector
  case value_1
    statements_1
  ...
  case value_n
    statements_n
  [ case else
    default_statements]
end select
```

selector is an expression which should evaluate as integral value. *values* can be literals, constants or expressions and *statements* can be any statements. The case else clause is optional.

First, the *selector* expression (condition) is evaluated. The *select case* statement then compares it against all available *values*. If the match is found, the *statements* following the match evaluate, and the *select case* statement terminates. In case there are multiple matches, the first matching statement will be executed. If none of the *values* matches the *selector*, then *default_statements* in the *case else* clause (if there is one) are executed.

Here is a simple example of the select case statement:

```
select case operator
  case "*"
    res = n1 * n2
  case "/"
    res = n1 / n2
  case "+"
    res = n1 + n2
  case "-"
    res = n1 - n2
  case else
    res = 0
    cnt = cnt + 1
end select
```

Also, you can group values together for a match. Simply separate the items by commas:

```
select case reg
  case 0
    opmode = 0
  case 1,2,3,4
    opmode = 1
  case 5,6,7
    opmode = 2
end select
```

Nested Case Statements

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

Iteration Statements (Loops)

Iteration statements let you loop a set of statements. There are three forms of iteration statements in mikroBasic for dsPIC30/33 and PIC24:

- for
- while
- do

You can use the statements break and continue to control the flow of a loop statement. break terminates the statement in which it occurs, while continue begins executing the next iteration of the sequence.

For Statement

The `for` statement implements an iterative loop and requires you to specify the number of iterations. The syntax of the for statement is:

```
for counter = initial_value to final_value [ step step_value]
  statements
next counter
```

counter is a variable being increased by *step_value* with each iteration of the loop. The parameter *step_value* is an optional integral value, and defaults to 1 if omitted. Before the first iteration, counter is set to *initial_value* and will be incremented until it reaches (or exceeds) the *final_value*. With each iteration, statements will be executed.

`initial_value` and `final_value` should be expressions compatible with *counter*; `statements` can be any statements that do not change the value of *counter*.

Note that the parameter *step_value* may be negative, allowing you to create a countdown.

Here is an example of calculating scalar product of two vectors, a and b, of length `n`, using the `for` statement:

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

Endless Loop

The `for` statement results in an endless loop if `final_value` equals or exceeds the range of the `counter`'s type.

While Statement

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

```
while expression
  statements
wend
```

`statements` are executed repeatedly as long as *`expression`* evaluates true. The test takes place before *`statements`* are executed. Thus, if *`expression`* evaluates false on the first pass, the loop does not execute.

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

```
s = 0
i = 0
while i < n
  s = s + a[ i] * b[ i]
  i = i + 1
wend
```

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

```
while TRUE
  ' ...
wend
```

Do Statement

The `do` statement executes until the condition becomes true. The syntax of the `do` statement is:

```
do
   statements
loop until expression
```

statements are executed repeatedly until *expression* evaluates true. *expression* is evaluated *after* each iteration, so the loop will execute *statements* at least once.

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

```
s = 0
i = 0
do
   s = s + a[ i] * b[ i]
   i = i + 1
loop until i = n
```

Jump Statements

A jump statement, when executed, transfers control unconditionally. There are five such statements in mikroBasic for dsPIC30/33 and PIC24:

- break
- continue
- exit
- goto
- gosub

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 inner-most loop (for, while, or do).

For example:

```
' Wait for CF card to be plugged; refresh every second
while true
  Lcd_Out(1, 1, "No card inserted")
  if Cf_Detect() = 1 then
    break
  end if
  Delay_ms(1000)
wend

' Now we can work with CF card ...
Lcd_Out(1, 1, "Card detected   ")
```

Continue Statement

You can use the continue statement within loops to "skip the cycle":

- continue statement in the for loop moves program counter to the line with keyword for
- continue statement in the while loop moves program counter to the line with loop condition (top of the loop),
- continue statement in the do loop moves program counter to the line with loop condition (bottom of the loop).

```
' continue jumps here
for i = ...
  ...
  continue
  ...
next i
```

```
' continue jumps here
while condition
  ...
  continue
  ...
wend
```

```
do
  ...
  continue
  ...
' continue jumps here
loop until condition
```

Exit Statement

The `exit` statement allows you to break out of a routine (function or procedure). It passes the control to the first statement following the routine call.

Here is a simple example:

```
sub procedure Proc1()
dim error as byte
  ... ' we're doing something here
  if error = TRUE then
    exit
  end if
  ... ' some code, which won't be executed if error is true
end sub
```

Note: If breaking out of a function, return value will be the value of the local variable *result* at the moment of exit.

Goto Statement

Use the `goto` statement to unconditionally jump to a local label — for more information, refer to Labels. The syntax of the `goto` statement is:

```
goto label_name
```

This will transfer control to the location of a local label specified by `label_name`. The goto line can come before or after the label.

Label and `goto` statement must belong to the same block. Hence it is not possible to jump into or out of a procedure or function.

You can use `goto` to break out from any level of nested control structures. Never jump into a *loop* or other structured statement, since this can have unpredictable effects.

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

```
for i = 0 to n
  for j = 0 to m
    ...
    if disaster
      goto Error
    end if
    ...
  next j
next i
 .
 .
 .
Error: ' error handling code
```

Gosub Statement

Use the `gosub` statement to unconditionally jump to a local label — for more information, refer to Labels. The syntax of the `gosub` statement is:

```
gosub label_name
...
label_name:
...
return
```

This will transfer control to the location of a local label specified by *`label_name`*. Also, the calling point is remembered. Upon encountering the `return` statement, program execution will continue with the next statement (line) after `gosub`. The `gosub` line can come before or after the label.

It is not possible to jump into or out of routine by means of gosub. Never jump *into* a loop or other structured statement, since this can have unpredictable effects.

Note: Like with `goto`, the use of `gosub` statement is generally discouraged. mikroBasic for dsPIC30/33 and PIC24 supports `gosub` only for the sake of backward compatibility. It is better to rely on functions and procedures, creating legible structured programs.

asm Statement

mikroBasic for dsPIC30/33 and PIC24 allows embedding assembly in the source code by means of the `asm` statement. Note that you cannot use numerals as absolute addresses for register variables in assembly instructions. You may use symbolic names instead (listing will display these names as well as addresses).

You can group assembly instructions with the `asm` keyword:

```
asm
   block of assembly instructions
end asm
```

mikroBasic comments are not allowed in embedded assembly code. Instead, you may use one-line assembly comments starting with semicolon.

If you plan to use a certain mikroBasic variable in embedded assembly only, be sure to at least initialize it (assign it initial value) in mikroBasic code; otherwise, the linker will issue an error. This is not applied to predefined globals such as PORTB.

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

```
program test

dim myvar as word

main:
  asm
    MOV #10, W0
    MOV W0, _myvar
  end asm
end.
```

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

```
myvar = 20
```

DIRECTIVES

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

The following directives are at your disposal:

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

Compiler Directives

Any line in source code with leading `#` is taken as a compiler directive. The initial `#` can be preceded or followed by whitespace (excluding new lines). The compiler directives are not case sensitive.

You can use conditional compilation to select particular sections of code to compile while excluding other sections. All compiler directives must be completed in the source file in which they begun.

Directives #DEFINE and #UNDEFINE

Use directive `#DEFINE` to define a conditional compiler constant ("flag"). You can use any identifier for a flag, with no limitations. No conflicts with program identifiers are possible because the flags have a separate name space. Only one flag can be set per directive.

For example:

```
#DEFINE extended_format
```

Use `#UNDEFINE` to undefine ("clear") previously defined flag.

Directives #IFDEF, #ELSEIF and #ELSE

Conditional compilation is carried out by the `#IFDEF` directive. `#IFDEF` tests whether a flag is currently defined or not; i.e. whether the previous `#DEFINE` directive has been processed for that flag and is still in force. The directive `#IFDEF` is terminated by the `#ENDIF` directive and can have any number of the `#ELSEIF` clauses and an optional `#ELSE` clause:

```
#IFDEF flag THEN
  block of code
[ #ELSEIF flag_1 THEN
  block of code 1
...
#ELSEIF flag_n THEN
  block of code n ]
[ #ELSE
  alternate block of code ]
#ENDIF
```

First, `#IFDEF` checks if `flag` is set by means of `#DEFINE`. If so, only *block of code* will be compiled. Otherwise, the compiler will check flags `flag_1 .. flag_n` and execute the appropriate *block of code i*. Eventually, if none of the flags is set, *alternate block of code* in `#ELSE` (if any) will be compiled.

`#ENDIF` ends the conditional sequence. The result of the preceding scenario is that only one section of code (possibly empty) is passed on for further processing. The processed section can contain further conditional clauses, nested to any depth; each `#IFDEF` must be matched with a closing `#ENDIF`.

Here is an example:

```
' Uncomment the appropriate flag for your application:
'#DEFINE resolution8
'#DEFINE resolution10
'#DEFINE resolution12
#IFDEF resolution8 THEN
  ... ' code specific to 8-bit resolution
#ELSEIF resolution10 THEN
  ... ' code specific to 10-bit resolution
#ELSEIF resolution12 THEN
  ... ' code specific to 12-bit resolution
#ELSE
  ... ' default code
#ENDIF
```

Predefined Flags

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

```
#IFDEF P30F4013 ' If dsPIC30F4013 MCU is selected
#IFDEF P30   ' P30F4013 and P30 flags will be automatically defined
```

Linker Directives

mikroBasic for dsPIC30/33 and PIC24 uses internal algorithm to distribute objects within memory. If you need to have a variable or routine at the specific predefined address, use the linker directives `absolute` and `org`.

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

Directive absolute

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

The `absolute` directive is appended to the declaration of a variable:

```
dim x as word absolute 0x32
' Variable x will occupy 1 word (16 bits) at address 0x32

dim y as longint absolute 0x34
' Variable y will occupy 2 words at addresses 0x34 and 0x36
```

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

```
dim  i as word absolute 0x42
' Variable i will occupy 1 word at address 0x42;

dim jj as longint absolute 0x40
' Variable will occupy 2 words at 0x40 and 0x42; thus,
' changing i changes jj at the same time and vice versa
```

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

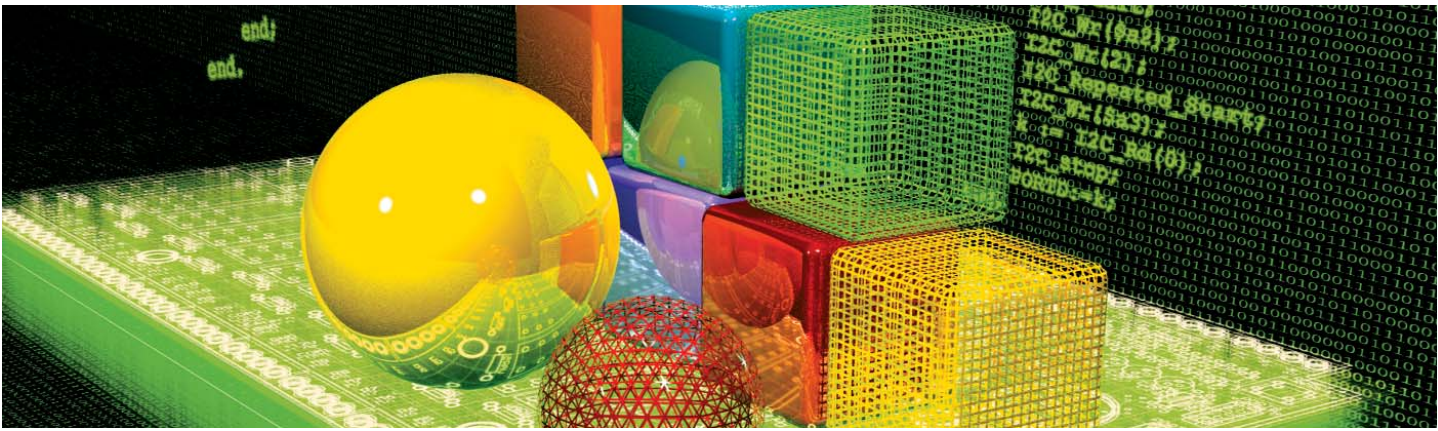
Directive org

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

```
sub procedure proc(dim par as word) org 0x200
' Procedure will start at the address 0x200;
...
end sub
```

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

CHAPTER 4



mikroBasic Libraries

mikroBasic provides a number of built-in and library routines which can help you develop your applications faster and easier. Hardware libraries for ADC, Advanced SPI Ethernet, Button, CAN, CANSPI, Compact Flash, DSP, Enhanced CAN, EEPROM, Flash Memory, Graphic LCD, I²C, Keypad, LCD Custom (4-bit interface), LCD8 Custom (8-bit interface), Manchester Code, Multi Media Card, OneWire, Port Expander, PS/2, PWM, PWM Motor, RS-485, Software I²C, Software SPI, Software UART, Sound, SPI, SPI Ethernet, SPI Graphic LCD, SPI LCD, SPI LCD8, Spi T6963C Graphic LCD, T6963C Graphic LCD, UART as well as utilities like Conversions, Math, Setjmp, String, Time, Util and Built-in Routines are included along with practical, ready-to-use code examples.

BUILT-IN ROUTINES

mikroBasic for dsPIC30/33 and PIC24 compiler provides a set of useful built-in utility functions. Built-in functions do not have any special requirements. You can use them in any part of your project.

`Vdelay_ms`, `Delay_Cyc` and `Get_Fosc_kHz` are actual Basic routines. Their sources can be found in the `delays.dbas` file located in the `uses` folder of the compiler.

```
Chr
Ord
SetBit
ClearBit
TestBit
Inc
Dec
Delay_us
Delay_ms
VDelay_ms_p30
VDelay_ms_p24_p33
Delay_Cyc
Clock_Khz
Clock_Mhz
Get_Fosc_kHz
```

Chr

Prototype	`sub function Chr(dim code as byte) as char`
Returns	Returns a character associated with the specified character code.
Description	Function returns a character associated with the specified character `code`. Numbers from 0 to 31 are the standard non-printable ASCII codes. This is an "inline" routine; the code is generated in the place of the call.
Example	`c = Chr(10)  ' returns the linefeed character`

Ord

Prototype	`sub function Ord(dim character as char) as byte`
Returns	ASCII code of the `character`.
Description	Function returns ASCII code of the `character`. This is an "inline" routine; the code is generated in the place of the call
Example	`c = Ord("A")  ' returns 65`

SetBit

Prototype	`sub procedure SetBit(dim byref register as word, dim rbit as word)`
Returns	Nothing.
Description	Function sets the bit `rbit` of `register`. Parameter `rbit` needs to be a variable or literal with value 0..15. For more information on register identifiers see Predefined Globals and Constants . This is an "inline" routine; the code is generated in the place of the call.
Example	`SetBit(PORTB, 2)  ' Set RB2`

ClearBit

Prototype	`sub procedure ClearBit(dim byref register as word, dim rbit as word)`
Returns	Nothing.
Description	Function clears the bit `rbit` of `register`. Parameter `rbit` needs to be a variable or literal with value 0..15. For more information on register identifier see Predefined Globals and Constants. This is an "inline" routine; the code is generated in the place of the call.
Example	`ClearBit(PORTC, 7)  ' Clear RC7`

TestBit

Prototype	`sub function TestBit(dim register, rbit as word) as word`
Returns	If the bit is set, returns 1, otherwise returns 0.
Description	Function tests if the bit `rbit` of `register` is set. If set, function returns 1, otherwise returns 0. Parameter rbit needs to be a variable or literal with value 0..15. For more information on register identifiers see Predefined Globals and Constants. This is an "inline" routine; the code is generated in the place of the call.
Example	`flag = TestBit(PORTE, 2)  ' 1 if RE2 is set, otherwise 0`

Inc

Prototype	`sub procedure Inc(dim byref par as longint)`
Returns	Nothing.
Description	Increases parameter `par` by 1.
Example	`p = 4` `Inc(p)  ' p is now 5`

Dec

Prototype	`sub procedure Dec(dim byref par as longint)`
Returns	Nothing.
Description	Decreases parameter `par` by 1.
Example	`p = 4` `Dec(p)  ' p is now 3`

Delay_us

Prototype	`sub procedure Delay_us(dim Time_In_us as word)`
Returns	Nothing.
Description	Creates a software delay in duration of `Time_In_us` microseconds. This is an "inline" routine; the code is generated in the place of the call, so the call doesn't count against the nested call limit. Parameters : - `Time_In_us`: delay time in microseconds. Valid values: constant values, range of applicable constants depends on the oscillator frequency
Example	`Delay_us(10)  ' Ten microseconds pause`

Delay_ms

Prototype	`sub procedure Delay_ms(dim Time_In_ms as word)`
Returns	Creates a software delay in duration of `Time_In_ms` milliseconds. This is an "inline" routine; the code is generated in the place of the call, so the call doesn't count against the nested call limit. Parameters : - `Time_In_ms`: delay time in milliseconds. Valid values: constant values, range of applicable constants depends on the oscillator frequency **Note**:For generating delays with variable as input parameter use the `Vdelay_ms` routine.
Description	Nothing.
Example	`Delay_ms(1000)  ' One second pause`

VDelay_ms_p30

Prototype	`sub procedure VDelay_ms_p30(dim Time_ms as word)`
Description	Creates a software delay in duration of `Time_ms` milliseconds. Generated delay is not as precise as the delay created by Delay_ms. Parameters : - `Time_ms`: delay time in milliseconds **Note**: `VDelay_ms_p30` is a library function rather than a built-in routine; it is presented in this topic for the sake of convenience. **Note**: `VDelay_ms_p30` can be used only with dsPIC30 family MCUs.

Example	`dim pause as word`
	`...`
	`VDelay_ms_p30(pause)  ' ~ one second pause`

VDelay_ms_p24_p33

Prototype	`sub procedure VDelay_ms_p24_p33(dim Time_ms as word)`
Description	Creates a software delay in duration of `Time_ms` milliseconds. Generated delay is not as precise as the delay created by Delay_ms. Parameters : - `Time_ms`: delay time in milliseconds **Note**: `VDelay_ms_p24_p33` is a library function rather than a built-in routine; it is presented in this topic for the sake of convenience. **Note**: `VDelay_ms_p24_p33` can be used only with dsPIC33 and PIC24 family MCUs.

Example	`dim pause as word`
	`...`
	`VDelay_ms_p24_p33(pause)  ' ~ one second pause`

Delay_Cyc

Prototype	`sub procedure Delay_Cyc(dim x as word, dim y as word)`
Returns	Nothing.
Description	Creates a delay based on MCU clock. Delay lasts for `x*16384` + y MCU clock cycles. Parameters : - `x`: NumberOfCycles divided by 16384 - `y`: remainder of the NumberOfCycles/16384 division **Note**: `Delay_Cyc` is a library function rather than a built-in routine; it is presented in this topic for the sake of convenience.
Example	`Delay_Cyc(1, 10)  ' 1x16384 + 10 = 16394 cycles pause`

Clock_Khz

Prototype	`sub function Clock_Khz() as longint`
Returns	Device clock in KHz, rounded to the nearest integer.
Description	Returns device clock in KHz, rounded to the nearest integer. This is an "inline" routine; the code is generated in the place of the call.
Example	`clk = Clock_Khz()`

Clock_Mhz

Prototype	`sub function Clock_Mhz() as word`
Returns	Device clock in MHz, rounded to the nearest integer.
Description	Returns device clock in MHz, rounded to the nearest integer. This is an "inline" routine; the code is generated in the place of the call.
Example	`clk = Clock_Mhz()`

Get_Fosc_kHz

Prototype	`sub function Get_Fosc_kHz() as longint`
Returns	Device clock in KHz.
Description	Function returns device clock in KHz, rounded to the nearest integer. **Note**: `Get_Fosc_kHz` is a library function rather than a built-in routine; it is presented in this topic for the sake of convenience.
Example	`clk = Get_Fosc_kHz()`

LIBRARY ROUTINES

mikroBasic for dsPIC30/33 and PIC24 provides a set of libraries which simplify the initialization and use of dsPIC30/33 and PIC24 and their modules:

Hardware Libraries

- ADC Library
- Advanced SPI Ethernet Library
- Button Library
- CAN Library
- CANSPI Library
- Compact Flash Library
- DSP Library
- Enhanced CAN Library
- EEPROM Library
- Flash Memory Library
- Graphic LCD Library
- I²C Library
- Keypad Library
- LCD Custom (4-bit interface) Library
- LCD8 Custom (8-bit interface) Library
- Manchester Code Library
- Multi Media Card Library
- OneWire Library
- Port Expander Library
- PS/2 Library
- PWM Library
- PWM Motor Library
- RS-485 Library
- Software I²C Library
- Software SPI Library
- Software UART Library
- Sound Library
- SPI Library
- SPI Ethernet Library
- SPI Graphic LCD Library
- SPI LCD Library
- SPI LCD8 Library
- Spi T6963C Graphic LCD Library
- T6963C Graphic LCD Library
- UART Library

Utilities

- Conversions Library
- Math Library
- Setjmp Library
- String Library
- Time Library
- Util Library

ADC Library

ADC (Analog to Digital Converter) module is available with a number of dsPIC30/33 and PIC24 MCU modules. ADC is an electronic circuit that converts continuous signals to discrete digital numbers. ADC Library provides a comfortable work with the module.

Library Routines

dsPIC30 and PIC24FJ Functions

```
ADC_Read
```

dsPIC33F and PIC24HJ Functions

```
Adc1_Read
Adc2_Read
Adc_Read
```

Adc_Read

Prototype	`sub function Adc_Read(dim channel as word) as word`
Returns	10-bit or 12-bit (MCU dependent) unsigned value from the specified `channel`.
Description	The function reads the specified analog channel input. The internal ADC dsPIC30F and PIC24FJ module is set to: - single channel conversion - unsigned integer data format - auto-convert - VRef+ : AVdd, VRef- : AVss - instruction cycle clock - conversion clock : 32*Tcy - auto-sample time : 31TAD Parameters : - `channel` represents the channel from which the analog value is to be acquired. Refer to the appropriate Datasheet for channel-to-pin mapping. **Note**: The function sets the appropriate bit in the ADPCFG registers in order to enable analog function of the selected pin.
Requires	The dsPIC30F and PIC24FJ MCU with built-in ADC module. Datasheet documentation for specific device should be consulted. Before using the function, be sure to configure the appropriate TRISx bits to configure pins as inputs.

Example	`dim k as word`
	`...`
	`k = Adc_Read(10)  'read analog value from channel 10`

Adc1_Read

Prototype	`sub function Adc1_Read(dim channel, AdcMode as word) as word`
Returns	10-bit or 12-bit (depending on selected mode) unsigned value from the specified channel.
Description	The function reads the specified analog channel input. This library works with the ADC1 module. The internal ADC dsPIC33FJ and PIC24HJ module is set to: - single channel conversion - unsigned integer data format - auto-convert - VRef+ : AVdd, VRef- : AVss - instruction cycle clock - conversion clock : 32*Tcy - auto-sample time : 31TAD Parameters : - `channel` represents the channel from which the analog value is to be acquired. Refer to the appropriate Datasheet for channel-to-pin mapping. - `AdcMode` represents ADC resolution. Valid values: ADC_10bit and ADC_12bit. These are library predefined constants, ADC_10bit=0 and ADC_12bit=1. **Note**: The function sets the appropriate bit in the ADPCFG registers in order to enable analog function of the selected pin.
Requires	The dsPIC33FJ and PIC24HJ MCU with built-in ADC1 module. Datasheet documentation for specific device should be consulted. Before using the function, be sure to configure the appropriate TRISx bits to designate pins as inputs.

Example	`dim k as word`
	`...`
	`k = Adc1_Read(10, ADC_10bit)`
	`'read analog value from channel 10 in 10-bit resolution`

Adc2_Read

Prototype	`sub function Adc2_Read(dim channel, AdcMode as word) as word`
Returns	10-bit or 12-bit (depending on selected mode) unsigned value from the specified `channel`.
Description	The function reads the specified analog channel input. This library works with the ADC2 module. The internal ADC dsPIC33FJ and PIC24HJ module is set to: - single channel conversion - unsigned integer data format - auto-convert - VRef+ : AVdd, VRef- : AVss - instruction cycle clock - conversion clock : 32*Tcy - auto-sample time : 31TAD Parameters : - `channel` represents the channel from which the analog value is to be acquired. Refer to the appropriate Datasheet for channel-to-pin mapping. - `AdcMode` represents ADC resolution. Valid values: ADC_10bit and ADC_12bit. These are library predefined constants, ADC_10bit=0 and ADC_12bit=1. **Note**: The function sets the appropriate bit in the ADPCFG registers to enable analog function of the selected pin.
Requires	The dsPIC33FJ and PIC24HJ MCU with built-in ADC2 module. Datasheet documentation for specific device should be consulted. Before using the function, be sure to configure the appropriate TRISx bits to designate pins as inputs.

Example	`dim k as word`
	`...`
	`k = Adc2_Read(10, ADC_10bit)`
	`'read analog value from channel 10 in 10-bit resolution`

Adc_Read

Prototype	`sub function Adc_Read(dim channel, AdcMode as word) as word`
Returns	10-bit or 12-bit (depending on selected mode) unsigned value from the specified `channel`.
Description	This function calls Adc1_Read. It is kept to maintain backwards compatibility with the previous versions of mikroBasic for dsPIC30/33 and PIC24.
Requires	The dsPIC33FJ and PIC24HJ MCU with built-in ADC1 module. Datasheet documentation for specific device should be consulted. Before using the function, be sure to configure the appropriate TRISx bits to designate pins as inputs.

Example	`dim k as word`
	`...`
	`k = Adc_Read(10, ADC_10bit)`
	`'read analog value from channel 10 in 10-bit resolution`

Library Example

dsPIC30 and PIC24FJ Functions

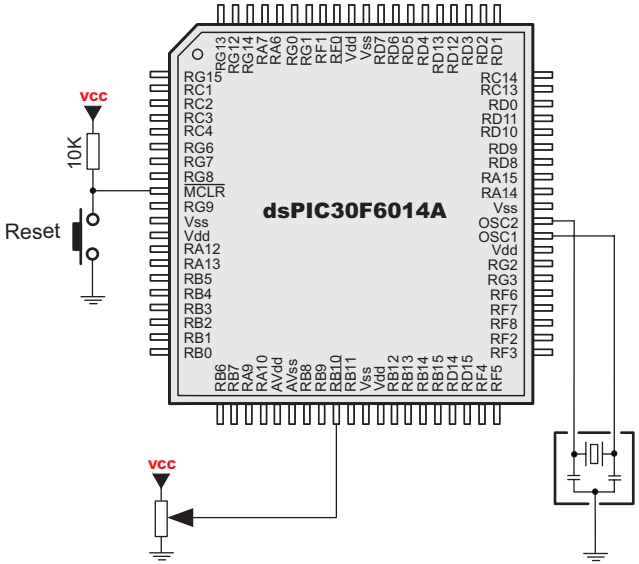
This code snippet reads analog value from channel 10 and sends its lower byte over UART1.

```
program ADC_test

dim adcRes as word
    txt     as string[ 5]

main:
  TRISB.10  = 1   ' set pin as input - needed for ADC to work

  Uart1_Init(9600)

  while true
    adcRes = Adc_Read(10)
    WordToStr(adcRes, txt)
    Uart1_Write_Text(txt)
    Delay_ms(50)
  wend

end.
```

This simple example reads analog value from channel 10 in 12-bit resolution and sends its lower byte via UART1.

```
program ADC_test

dim k as word

main:
  TRISB.10  = 1   ' set pin as input - needed for ADC to work

  Uart1_Init(9600)

  while TRUE
    k = Adc_Read(10, ADC_12bit)
    Uart1_Write_Char(k)
    Delay_ms(50)
  wend

end.
```

Hardware Connection



Advanced SPI Ethernet Library

This library is designed to simplify handling of the underlying hardware (ENC28J60). However, a certain level of knowledge about the Ethernet and Ethernet-based protocols (ARP, IP, TCP/IP, UDP/IP, ICMP/IP) is expected from the user. The Ethernet is a high–speed and versatile protocol, but not a simple one. Once you get used to it, you will make your favorite dsPIC30/33 and PIC24 available to a much broader audience than you could do with the RS232/485 (UART) or CAN.

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

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

Note: This library uses TIMER1 for timeout logic.

Library Routines

Ethernet Initialization

```
EthSetMACAddr
EthSetIPAddr
EthSetIPMask
EthSetGateWayAddr
EthInit
```

Media Access Control Layer (MAC)

```
MACInit
MACIsTxReady
MACGetHeader
MACGet
MACGetArray
MACDiscardRx
MACPutHeader
MACPut
MACPutArray
MACFlush
MACDiscardTx
MACSetTxBuffer
MACSetRxBuffer
MACReserveTxBuffer
MACGetFreeRxSize
MACSetDuplex
```

Address Resolution Protocol (ARP and ARPTask)

```
ARPGet
ARPPut
ARPInit
```

Internet Protocol (IP)

```
IPIsTxReady
IPSetTxBuffer
IPPutHeader
IPPutArray
IPGetHeader
IPGetArray
IPSetRxBuffer
```

Internet Control Message Protocol(ICMP)

```
ICMPIsTxReady
ICMPPut
ICMPGet
```

Transmission Control Protocol (TCP)

```
TCPInit
TCPListen
TCPConnect
TCPIsConnected
TCPDisconnect
TCPIsPutReady
TCPPut
TCPFlush
TCPIsGetReady
TCPGet
TCPGetArray
TCPDiscard
TCPProcess
TCPTick
```

User Datagram Protocol (UDP)

```
UDPInit
UDPOpen
UDPClose
UDPIsPutReady
UDPPut
UDPFlush
UDPIsGetReady
UDPGet
UDPDiscard
UDPProcess
UDPWrite
UDPRead
UDPOpenSocket
```

Other

```
StackInit
StackTask
HTTPInit
HTTPServer
```

EthSetMACAddr

Prototype	`sub procedure EthSetMACAddr(dim MACByte1, MACByte2, MACByte3, MACByte4, MACByte5, MACByte6 as byte)`
Returns	Nothing.
Description	This routine sets MAC address.
Requires	Nothing.
Example	`' Sets MAC address` `EthSetMACAddr(0, 2, 4, 6, 8, 10)`

EthSetIPAddr

Prototype	`sub procedure EthSetIPAddr(dim IpByte1, IpByte2, IpByte3, IpByte4 as byte)`
Description	This routine sets IP address.
Requires	Nothing.
Example	`' Sets IP address` `EthSetIPAddr(192,168,20,1)`

EthSetIPMask

Prototype	`sub procedure EthSetIPMask(dim IPMaskByte1, IPMaskByte2, IPMaskByte3, IPMaskByte4 as byte)`
Description	This routine sets IP address mask.
Requires	Nothing.
Example	`' Sets address mask` `EthSetIPMask(255,255,255,0)`

EthSetGateWayAddr

Prototype	`sub procedure EthSetGateWayAddr(dim GateWayByte1, GateWayByte2, GateWayByte3, GateWayByte4 as byte)`
Description	This routine sets Gateway IP address.
Requires	Nothing.
Example	`' Sets Gateway IP address` `EthSetGateWayAddr(192,168,20,1)`

EthInit

Prototype	`sub procedure EthInit(dim byref RstPort as word, dim RstPin as word, dim byref CSPort as word, dim CSPin as word)`
Description	This routine initializes NIC and global variable needed for Ethernet connection.
Requires	EthSetMACAddr, EthSetIPAddr, EthSetIPMask need to be set before calling this routine. Calling EthSetGateWayAddr before this routine is optional.
Example	`' Initializes Ethernet connection` `EthInit(PORTF,0,PORTF,1)`

MACInit

Prototype	`sub procedure MACInit(dim byref RstPort as word, dim RstPin as word, dim byref CSPort as word, dim CSPin as word)`
Description	This function initializes MAC layer. It initializes internal buffers and resets the NIC to a known state. All pending transmissions and receptions are discarded.
Requires	Nothing.
Example	`' Initialize MAC Module` `MACInit(PORTF,0,PORTF,1)`

MACIsTxReady

Prototype	`sub function MACIsTxReady() as byte`
Returns	(<>0) - If at least one MAC transmit buffer is empty. (=0) - If all MAC transmit buffers are full.
Description	This function indicates whether at least one MAC transmit buffer is empty or not.
Requires	Nothing.

Example	`' Check MAC transmit readiness...`
	`if (MACIsTxReady() <> 0) then`
	
	`' Transmit buffer is empty, transmit a message.`
	`...`
	`end if`

MACGetHeader

Prototype	`sub function MACGetHeader(dim byref remote as MAC_ADDR, dim byref type as byte) as byte`
Returns	(<>0) : If a data packet is received and found to be valid. All parameters are populated. (=0) : If no data packet is received or found to be invalid.
Description	This function checks the MAC receive buffer; if any packet is found, it returns the remote host and data packet information. `Remote` specifies Remote MAC address and `type` represents Data packet type. Possible values for type parameter are: MAC_IP (An IP data packet is received), MAC_ARP (An ARP data packet is received) MAC_UNKNOWN (An unknown or unsupported data packet is received).
Requires	Nothing.

Example	`' Get possible data packet info.`
	`if(MACGetHeader(RemoteNodeMAC, PacketType) <> 0) then`
	`  ' A packet is received, process it.`
	`  ...`
	`  ' Once done, discard it.`
	`  MACDiscardRx()`
	`  ...`
	`end if`

MACGet

Prototype	`sub function MACGet() as byte`
Returns	Data byte.
Description	This function returns the next byte from an active transmit or receive buffer.
Requires	MACGetHeader, MACPutHeader, MACSetRxBuffer or MACSetTxBuffer must be called.

Example	`' Get possible data packet info.`
	`if (MACGetHeader(RemoteNode, PacketType) <> 0) then`
	`  ' A packet is received, process it.`
	`  data = MACGet()`
	`  ...`
	`  ' When done, discard it.`
	`  MACDiscardRx()`
	`  ...`
	`end if`

MACGetArray

Prototype	`sub function MACGetArray(dim byref val as byte, dim len as word) as word`
Returns	Total number of bytes fetched.
Description	This function fetches an array of bytes from the active transmit or receive buffer. val represents pointer to a byte array and len represents number of bytes to fetch.
Requires	MACGetHeader, MACPutHeader,MACSetRxBuffer or MACSetTxBuffer must be called.

Example	`' Get possible data packet info.`
	`if (MACGetHeader(RemoteNode, PacketType) <> 0) then`
	`  ' A packet is received, process it.`
	`  actualCount = MACGetArray(data, count)`
	`  ...`
	`end if`

MACDiscardRx

Prototype	`sub procedure MACDiscardRx()`
Returns	Nothing.
Description	This function discards the active receive buffer data and marks that buffer as free.
Requires	Nothing.

Example	`' Get possible data packet info.`
	`if (MACGetHeader(RemoteNode, PacketType) <> 0) then`
	`  ' A packet is received, process it.`
	`  actualCount = MACGetArray(data, count)`
	`  ...`
	`  ' Done processing it. Discard it.`
	`  MACDiscardRx()`
	`  ...`
	`end if`

MACPutHeader

Prototype	`sub procedure MACPutHeader(dim byref remote as MAC_ADDR, dim type as byte, dim dataLen as word)`
Returns	Nothing.
Description	This function assembles the MAC header and loads it to an active transmit buffer. `remote` - Remote node MAC address, `type` - Type of data packet being sent. Possible values for this parameter are: MAC_IP(An IP data packet is to be transmitted) and MAC_ARP(An ARP data packet is to be transmitted), `data` - Number of bytes for this packet, including IP header.
Requires	Nothing.

Example	`' Check to see if at least one transmit buffer is empty`
	`if (MACIsTxReady() <> 0) then`
	`  ' Assemble IP packet with total IP packet size of 100 bytes`
	`  ' including IP header.`
	`  MACPutHeader(RemoteNodeMAC, MAC_IP, 100)`
	`  ...`
	`end if`

MACPut

Prototype	`sub procedure MACPut(dim val as byte)`
Returns	Nothing.
Description	This function loads the given data byte into an active transmit or receive buffer. `val` - Data byte to be written.
Requires	MACGetHeader, MACPutHeader, MACSetRxBuffer or MACSetTxBuffer must be called.

Example	`' Check to see if at least one transmit buffer is empty`
	`if (MACIsTxReady() <> 0) then`
	`  ' Assemble IP packet with total IP packet size of 100 bytes`
	`  ' including IP header.`
	`  MACPutHeader(RemoteNodeMAC, MAC_IP, 100)`
	`  ' Now put the actual IP data bytes`
	`  MACPut($55)`
	`  ...`
	`end if`

MACPutArray

Prototype	`sub procedure MACPutArray(dim byref val as byte, dim len as word)`
Returns	Nothing.
Description	This function writes an array of data bytes into an active transmit or receive buffer. `val` - Data bytes to be written. `len` - Total number of bytes to write.
Requires	MACGetHeader, MACPutHeader, MACSetTxBuffer or MACSetRxBuffer must be called.

Example	`' Check to see if at least one transmit buffer is empty`
	`if (MACIsTxReady() <> 0) then`
	`  ' Assemble IP packet with total IP packet size of 100 bytes`
	`  ' including IP header.`
	`  MACPutHeader(RemoteNodeMAC, MAC_IP, 100)`
	`  ' Now put the actual IP data bytes`
	`  MACPut($55)`
	`  MACPutArray(data, count)`
	`  ...`
	`end if`

MACFlush

Prototype	`sub procedure MACFlush()`
Description	This function marks active transmit buffer as ready for transmission.
Requires	MACPutHeader or MACSetTxBuffer must be called.

Example	`' Check to see if at least one transmit buffer is empty`
	`if (MACIsTxReady() <> 0) then`
	`  ' Assemble IP packet with total IP packet size of 100 bytes`
	`  ' including IP header.`
	`  MACPutHeader(RemoteNodeMAC, MAC_IP, 100)`
	`  ' Put the actual IP data bytes`
	`  MACPut($55)`
	`  MACPutArray(data, count)`
	`  ...`
	`  ' Now transmit it.`
	`  MACFlush()`
	`  ...`
	`end if`

MACDiscardTx

Prototype	`sub procedure MACDiscardTx(dim buffer as BUFFER)`
Description	This function discards given transmit buffer content and marks it as free.
Requires	Nothing.

Example	`' Check to see if at least one transmit buffer is empty`
	`if (MACIsTxReady() <> 0) then`
	`  ' Assemble IP packet with total IP packet size of 100 bytes`
	`  ' including IP header.`
	`  MACPutHeader(RemoteNodeMAC, MAC_IP, 100)`
	`  ' Get current transmit buffer`
	`  buffer = MACGetTxBuffer()`
	`  ' Reserve it.`
	`  MACReserveTxBuffer(Buffer)`
	`  ' Put the actual IP data bytes`
	`  ...`
	`  ' Now transmit it.`
	`  MACFlush()`
	`  ' No longer need this buffer`
	`  MACDiscardTx(buffer)`
	`  ...`
	`end if`

MACSetRxBuffer

Prototype	`sub procedure MACSetRxBuffer(dim offset as word)`
Description	This function sets the access location for the active receive buffer. `offset` - Location (with respect to beginning of buffer) where next access is to occur.
Requires	Nothing.

Example	`' Get possible data packet info.`
	`if (MACGetHeader(RemoteNode, PacketType) <> 0) then`
	`  ' A packet is received, process it.`
	`  actualCount = MACGetArray(data, count)`
	`  ...`
	`  ' Fetch data beginning at offset 20`
	`  MACSetRxBuffer(20)`
	`  data = MACGet()`
	`  ...`
	`end if`

MACSetTxBuffer

Prototype	`sub procedure MACSetTxBuffer(dim buffer as BUFFER, dim offset as word)`
Description	This function sets the access location for a given transmit buffer and makes that transmit buffer active. `buffer` - A transmit buffer where this access offset will be applied. `offset` - Location (with respect to beginning of buffer) where next access is to occur.
Requires	Nothing.

Example	`' Check to see if at least one transmit buffer is empty`
	`if (MACIsTxReady() <> 0) then`
	`  ' Assemble IP packet with total IP packet size of 100 bytes`
	`  ' including IP header.`
	`  MACPutHeader(RemoteNodeMAC, MAC_IP, 100)`
	`  ' Get current transmit buffer`
	`  buffer = MACGetTxBuffer()`
	`  ' Put the actual IP data bytes`
	`  ...`
	`  ' Calculate the checksum of data packet`
	`  ' that is being transmitted`
	`  ...`
	`  ' Now update the checksum in this packet.`
	`  ' To update the checksum, set transmit`
	`  ' buffer access to checksum`
	`  MACSetTxBuffer(buffer, checksumLocation)`
	`  ...`
	`  ' Now transmit it.`
	`  MACFlush()`
	`  ...`
	`end if`

MACReserveTxBuffer

Prototype	`sub procedure MACReserveTxBuffer(dim buffer as BUFFER)`
Description	This function reserves a given transmit buffer and marks it as unavailable. This function is useful for TCP layer where a message will be queued until it is correctly acknowledged by remote host. `buffer` - A transmit buffer to reserve. This value must be a valid transmit buffer identifier as returned by the MACGetTxBuffer function.
Requires	Nothing.
Example	

```
' Check to see if at least one transmit buffer is empty
if (MACIsTxReady() <> 0) then
  ' Transmit IP packet with total IP packet size of 100 bytes
  ' including IP header.
  MACPutHeader(RemoteNodeMAC, MAC_IP, 100)
  ' Get current transmit buffer
  buffer = MACGetTxBuffer()
  ' Reserve it, to be discarded when ACK is received.
  MACReserveTxBuffer(buffer)
  ' Put the actual IP data bytes
  ...
  ' Calculate the checksum of data packet
  ' that is being transmitted
  ...
  ' Now update the checksum in this packet.
  ' To update the checksum, set transmit buffer
  ' access to checksum
  MACSetTxBuffer(buffer, checksumLocation)
  ...
  ' Now transmit it.
  MACFlush()
  ...
end if
```

MACGetFreeRxSize

Prototype	`sub function MACGetFreeRxSize() as word`
Returns	Total number of bytes available for future data packets.
Description	This function returns total receive buffer size available for future data packets.
Requires	Nothing.
Example	`' Get available receive buffer size` `freeRxBuffer = MACGetFreeRxSize()`

MACSetDuplex

Prototype	`sub procedure MACSetDuplex(dim DuplexState as DUPLEX)`
Returns	Nothing.
Description	This routine sets FULL-DUPLEX or HALF-DUPLEX communication mode. Input value can be **1** for FULL-DUPLEX, 0 for HALF-DUPLEX and **2** for NIC defined communication mode.
Requires	Nothing.
Example	`MACSetDuplex(1) ' FULL-DUPLEX communication is set`

ARPGet

Prototype	`sub function ARPGet(dim byref remote as NODE_INFO, dim byref opCode as byte) as byte`
Returns	(<>0) - If a valid ARP packet that was addressed to a local host was fetched; remote and opCode contain valid values. (=0) - Either unknown ARP code was fetched or this packet was not addressed to a local host.
Description	This function fetches complete ARP packet and returns necessary information. `remote` - Remote node information such as MAC and IP addresses. `opCode` - ARP code. Possible values for this parameter are: **ARP_REPLY** ("ARP Reply" packet is received), **ARP_REQUEST** ("ARP Request" packet is received), **ARP_UNKNOWN** (An unknown ARP packet is received).
Requires	MACGetHeader is already called AND Received MAC packet type = MAC_ARP

Example	`' If MAC packet is received...`
	`if (MACGetHeader(RemoteNode, PacketType) <> 0) then`
	`' If this is an ARP packet, fetch it.`
	`  if (PacketType = MAC_ARP) then`
	`  ' This is ARP packet.`
	`  ARPGet(RemoteNode, ARPCode)`
	`  ...`

ARPPut

Prototype	`sub procedure ARPPut(dim byref remote as NODE_INFO, dim opCode as byte)`
Description	This function loads MAC buffer with valid ARP packet. `remote` - Remote node information such as MAC and IP addresses. `opCode` - ARP code. Possible values for this parameter are: **ARP_REPLY** (Transmit this packet as "ARP Reply"), **ARP_REQUEST** (Transmit this packet as "ARP Request").
Requires	ARPIsTxReady = TRUE

Example	`' Check to see if transmit buffer is available`
	`if (ARPIsTxReady() <> 0) then`
	`  ' Transmit it`
	`  ARPPut(RemoteNode, ARP_REQUEST)`
	`  ...`

ARPInit

Prototype	`sub procedure ARPInit()`
Description	This function initializes the ARPTask state machine and prepares it to handle ARP requests and replies.
Requires	Nothing.

Example	`' Initialize ARPTask`
	`ARPInit()`
	`...`

IPIsTxReady

Prototype	`sub function IPIsTxReady() as byte`
Returns	(<>0) - If there is at least one transmit buffer empty. (=0) - If there is no empty transmit buffer.
Description	This is a macro that calls MACIsTxReady in turn.
Requires	Nothing.

Example	`' If IP transmit buffer is ready, transmit IP packet`
	`if (IPIsTxReady() <> 0) then`
	`  ' Assemble IP packet.`
	`  IPPutHeader(Remote, MAC_TCP, IPPacketLen)`
	`  ...`

IPSetTxBuffer

Prototype	`sub procedure IPSetTxBuffer(dim buffer as BUFFER, dim offset as word)`
Returns	Nothing.
Description	This is a macro to allow higher level layer set transmit buffer access pointer. This macro takes IP header into account before calling MACSetTxBuffer. `buffer` - Transmit buffer identifier whose access pointer is to be set. `offset` - An offset with respect to IP Data.
Requires	Nothing.

Example	`' If IP transmit buffer is ready, transmit IP packet`
	`if (IPIsTxReady() <> 0) then`
	`  ' Assemble IP packet.`
	`  IPPutHeader(Remote, MAC_TCP, IPPacketLen)`
	`  ' Get current transmit buffer id.`
	`  buffer = MACGetTxBuffer()`
	`  ' Load transmit data`
	`  ...`
	`  ' Calculate checksum checkHi:checkLo`
	`  ...`
	`  ' Update the checksum.`
	`  IPSetTxBuffer(buffer, checkLocation)`
	`  MACPut (checkHi)`
	`  MACPut (checkLo)`
	`  ...`

IPPutHeader

Prototype	`sub function IPPutHeader(dim byref remote as NODE_INFO, dim protocol as byte, dim len as word) as word`
Returns	Nothing.
Description	This function assembles a valid IP header and loads it into active transmit buffer. `remote` - Remote node information such as MAC and IP addresses. `protocol` - Protocol to use for this data packet. Possible values for this parameter are: IP_PROT_ICMP (Assemble this packet as ICMP), IP_PROT_TCP (Assemble this packet as TCP segment), IP_PROT_UDP (Assemble this packet as UDP segment). `len` - Total length of IP data bytes, excluding IP header.
Requires	IPIsTxReady = TRUE

Example	`' Check to see if transmit buffer is available`
	`if (IPIsTxReady() <> 0) then`
	`  ' Load the header`
	`  IPPutHeader(RemoteNode, IP_PROT_ICMP, ICMP_HEADER_SIZE+dataLen)`
	`  ' Load ICMP data`
	`  IPPutArray(ICMPData, dataLen)`
	`  ' Mark it as ready to be transmitted`
	`  MACFlush()`
	`  ...`

IPPutArray

Prototype	`sub procedure IPPutArray(dim byref buffer as byte, dim len as word)`
Returns	Nothing.
Description	This macro loads an array of bytes into the active transmit buffer. `buffer` - Data array to be loaded. `len` - Total number of items in data array.
Requires	IPIsTxReady = TRUE

Example	`' Check to see if transmit buffer is available`
	`if (IPIsTxReady() <> 0) then`
	`  ' Load the header`
	`  IPPutHeader(RemoteNode, IP_PROT_ICMP, ICMP_HEADER_SIZE+dataLen)`
	`  ' Load ICMP data`
	`  IPPutArray(ICMPData, dataLen)`
	`  ' Mark it as ready to be transmitted`
	`  MACFlush()`
	`  ...`

IPGetHeader

Prototype	`sub function IPGetHeader(dim byref localIP as IP_ADDR, dim byref remote as NODE_INFO, dim byref protocol as byte, dim byref len as word) as byte`
Returns	(<>0) - A valid IP packet was received. Remote IP address, packet protocol and packet length parameters are populated. (=0) - An invalid IP packet was received. Parameters are not populated.
Description	This function fetches the IP header from the active transmit buffer and validates it. `localIP` - Local node information such as MAC and IP addresses, `remote` - Remote node information such as MAC and IP addresses, `protocol` - Protocol associated with this IP packet. Possible values for this parameter are: IP_PROT_ICMP (This is an ICMP packet), IP_PROT_TCP (This is a TCP packet), IP_PROT_UDP (This is a UDP packet), all others Unknown protocol. `len` - Total length of IP data in this packet.
Requires	MACGetHeader = TRUE

Example	`' Check to see if any packet is ready`
	`if (MACGetHeader(RemoteMACAddr, PacketType) <> 0) then`
	`' Check what kind of protocol it is`
	`  if (PacketType = MAC_IP) then`
	`  ' This is IP packet. Fetch it.`
	`  IPGetHeader(Local, Remote, IPProtocol, IPLen)`
	`  ' Process this IP packet.`
	`  ...`
	`  ' When done processing this packet, discard it`
	`  MACDiscardRx()`
	`  end if`
	`else`
	`  ' This is not an IP packet. Handle it`
	`  ...`

IPGetArray

Prototype	`sub function IPGetArray(dim byref val as byte, dim len as word) as word`
Returns	Total number of bytes fetched.
Description	This macro fetches an array of bytes from an active transmit or receive buffer. `val` - data buffer. `len` - Number of bytes to fetch.
Requires	IPGetHeader, IPPutHeader, IPSetRxBuffer or IPSetTxBuffer must be called.

Example	`' Check to see if any packet is ready`
	`if (MACGetHeader(RemoteMACAddr, PacketType) <> 0) then`
	`' Check what kind of protocol it is`
	`  if (PacketType = MAC_IP) then`
	`  ' This is IP packet. Fetch it.`
	`  IPGetHeader(Remote, IPProtocol, IPLen)`
	`  ' Get 20 bytes of data`
	`  IPGetArray(IPData, 20)`
	`  ...`
	`  ' When done processing this packet, discard it`
	`  MACDiscardRx()`
	`  end if`
	`else`
	`  ' This is not an IP packet. Handle it`
	`  ...`

IPSetRxBuffer

Prototype	`sub procedure IPSetRxBuffer(dim offset as word)`
Returns	Nothing.
Description	This macro allows a higher level layer to set the receive buffer access pointer. It takes the IP header into account before calling MACSetRxBuffer. `offset` - An offset with respect to IP Data.
Requires	Nothing.
Example	

```
' Check to see if any packet is ready
if (MACGetHeader(RemoteMACAddr, PacketType) <> 0) then
  ' Check what kind of protocol it is
  if (PacketType = MAC_IP) then
    ' This is IP packet. Fetch it.
    IPGetHeader(Remote, IPProtocol, IPLen)
    ' Fetch 20th byte within IP data.
    IPSetRxBuffer(20)
    data = MACGet()
    ...
    // When done processing this packet, discard it
    MACDiscardRx()
  end if
else
  ' This is not an IP packet. Handle it
  ...
```

ICMPIsTxReady

Prototype	`sub function ICMPIsTxReady() as byte`
Returns	(<>0) - If there is at least one transmit buffer empty. (=0) - If there is no empty transmit buffer.
Description	This macro determines if at least one transmit buffer is empty.
Requires	Nothing.

Example	`' If IP transmit buffer is ready, transmit IP packet`
	`if (ICMPIsTxReady() <> 0) then`
	`  ' Transmit ICMP packet.`
	`  ...`

ICMPPut

Prototype	`sub procedure ICMPPut(dim byref remote as NODE_INFO, dim code as ICMP_CODE, dim byref data as byte, dim len as byte, dim id, seq as word)`
Returns	Nothing.
Description	This function assembles a valid ICMP packet and transmits it. `remote` - Remote node information such as MAC and IP addresses, `code` - ICMP code to be used for this ICMP packet. Possible values for this parameter are: ICMP_ECHO_REPLY (This is an ICMP Echo reply packet), ICMP_ECHO_REQUEST (This is an ICMP Echo request packet). `data` - ICMP data. `len` - ICMP data length. `id` - ICMP packet identifier. `seq` - ICMP packet sequence number.
Requires	IPIsTxReady = TRUE

Example	`' Check to see if transmit buffer is available`
	`if (ICMPIsTxReady() <> 0) then`
	`  ' Transmit ICMP packet.`
	`  ICMPPut(RemoteNode, ICMP_ECHO_REPLY, data, datalen, id, seq)`
	`  ' Done. ICMP is put into transmit queue.`
	`  ...`

ICMPGet

Prototype	`sub procedure ICMPGet(dim byref remote as NODE_INFO, dim byref code as ICMP_CODE, dim byref data, len as byte, dim byref id, seq as word)`
Returns	(<>0) - A valid ICMP packet was received. All parameters are populated. (=0) - An invalid ICMP packet was received. Parameters are not populated.
Description	This function fetches the ICMP header from the active transmit buffer and validates it. `remote` - Remote node information such as MAC and IP addresses. `code` - ICMP code for received ICMP packet. Possible values for this parameter are: ICMP_ECHO_REPLY (An ICMP Echo reply packet is received), ICMP_ECHO_REQUEST (An ICMP Echo request packet is received), for all others (An unknown/unsupported packet is received). `data` - ICMP data. `len` - ICMP data length. `id` - ICMP packet identifier. `seq` - ICMP packet sequence number.
Requires	IPGetHeader = TRUE PacketType = IP_PROT_ICMP

Example	`' Check to see if any packet is ready`
	`if (IPGetHeader(Remote, IPProtocol, IPLen) <> 0) then`
	`  ' Check what kind of protocol it is`
	`  if (IPProtocol = IP_PROT_ICMP) then`
	`  ' This is ICMPP packet. Fetch it.`
	`  ICMPGet(ICMPCode, data, dataLen, id, seq)`
	`  ' Process this ICMP packet.`
	`  ...`
	`  ' When done processing this packet, discard it`
	`  MACDiscardRx()`
	`  end if`
	`else`
	`  ' This is not an ICMP packet. Handle it`
	`  ...`

TCPInit

Prototype	`sub procedure TCPInit()`
Description	This function initializes the TCP state machine and prepares it for multiple TCP connections.
Requires	Nothing.
Example	`' Initialize TCP` `TCPInit()`

TCPListen

Prototype	`sub function TCPListen(dim port as TCP_PORT) as TCP_SOCKET`
Returns	A valid socket identifier if there is at least one free socket. INVALID_SOCKET if there is no socket available.
Description	This function assigns one of the available sockets to listen on given TCP port. `port` - TCP Port number on which to listen.
Requires	Nothing.

Example	`...`
	`httpSocket = TCPListen(80)`
	`If (httpSocket = INVALID_SOCKET) then`
	`' Socket is not available`
	`' Return error.`
	`...`

TCPConnect

Prototype	`sub function TCPConnect(dim byref remote as NODE_INFO, dim port as TCP_PORT) as TCP_SOCKET`
Returns	A valid socket identifier if there is at least one free socket. INVALID_SOCKET if there is no socket available.
Description	This function initiates a connection request to a remote host on a given remote port. `remote` - Remote host that needs to be connected. `port` - TCP Port number on remote host to connect to.
Requires	Nothing.

Example	`...`
	`ftpSocket = TCPConnect(RemoteNode, 21)`
	`if (ftpSocket = INVALID_SOCKET) then`
	`  ' Socket is not available`
	`  ' Return error.`
	`  ...`

TCPIsConnected

Prototype	`sub function TCPIsConnected(dim socket as TCP_SOCKET) as byte`
Returns	(<>0) If given socket is connected to remote host. (=0) If given socket is not connected to remote host.
Description	This function determines whether a given socket is connected to remote host or not. `socket` - Socket identifier for which the connection is to be checked.
Requires	Nothing.

Example	`ftpSocket = TCPConnect(RemoteNode, 21)`
	`if (ftpSocket = INVALID_SOCKET) then`
	`' Socket is not available`
	`' Return error.`
	`end if`
	`else smState = SM_CONNECT_WAIT`
	
	`' Wait for connection...`
	`if (TCPIsConnected(ftpSocket) <> 0) then`
	`smState = SM_CONNECTED`
	`...`

TCPDisconnect

Prototype	`sub procedure TCPDisconnect(dim socket as TCP_SOCKET)`
Returns	Nothing.
Description	This function requests remote host to disconnect.
Requires	TCPIsPutReady = TRUE

Example	`' Disconnect`
	`TCPDisconnect(ftpSocket)`
	`...`

TCPIsPutReady

Prototype	`sub function TCPIsPutReady(dim socket as TCP_SOCKET) as byte`
Returns	(<>0) - If given socket is ready to transmit. (=0) - If given socket is not connected or there is no transmit buffer ready.
Description	This function determines if a socket is ready to transmit. A socket is ready to transmit when it is connected to a remote host and when its transmit buffer is empty. `socket` - Socket identifier that needs to be checked.
Requires	Nothing.

Example	`if (TCPIsPutReady(ftpSocket) <> 0) then`
	`' Send data`
	`...`

TCPPut

Prototype	`sub function TCPPut(dim socket as TCP_SOCKET, dim data as byte) as byte`
Returns	(<>0) - If a given data byte was successfully loaded into the transmit buffer and there is room for more data. (=0) - If a given data byte was successfully loaded into the transmit buffer and there is no room for more data.
Description	This function loads a data byte into the transmit buffer for a given socket. socket - Socket identifier that needs to be checked. `data` - Data byte to be loaded.
Requires	TCPIsPutReady = TRUE

Example	`...`
	`if (TCPIsPutReady(ftpSocket) <> 0) then`
	`' Send data`
	`TCPPut(ftpSocket, dataByte)`
	`...`

TCPFlush

Prototype	`sub procedure TCPFlush(dim socket as TCP_SOCKET)`
Returns	Nothing.
Description	This function marks given socket transmit buffer as ready to be transmitted. socket - Socket identifier that needs to be transmitted.
Requires	Nothing.

Example	`...`
	`if (TCPIsPutReady(ftpSocket) <> 0) then`
	`' Send data`
	`TCPPut(ftpSocket, dataByte)`
	`...`
	`' Now transmit it.`
	`TCPFlush(ftpSocket)`
	`...`

TCPIsGetReady

Prototype	`sub function TCPIsGetReady(dim socket as TCP_SOCKET) as byte`
Returns	Nothing.
Description	(<>0) - If given socket contains received data. (=0) - If given socket does not contain any data.
Requires	This function determines if the given socket contains received data. `socket` - Socket identifier that needs to be transmitted.

Example	`...`
	`' Fetch data`
	`if (TCPIsGetReady(httpSocket) <> 0) then`
	`' Fetch data`
	`...`

TCPGet

Prototype	`sub function TCPGet(dim socket as TCP_SOCKET, dim byref data as byte) as byte`
Returns	(<>0) - If a byte is read. (=0) - If no byte is read.
Description	This function fetches one data byte from a given socket receive buffer. `socket` - Socket identifier that needs to be fetched. `data` - Data byte that is read.
Requires	TCPIsGetReady = TRUE

Example	`...`
	`' Fetch data`
	`if (TCPIsGetReady(httpSocket) <> 0) then`
	`' Fetch data`
	`TCPGet(httpSocket, dataByte)`
	`...`

TCPGetArray

Prototype	`sub function TCPGetArray(dim socket as TCP_SOCKET, dim byref data as byte[1024], dim count as word) as word`
Returns	Total number of data bytes read.
Description	This function fetches a data array from a given socket receive buffer. `socket` - Socket identifier that needs to be fetched. `data` - Data array that is read. `count` - Total number of bytes to read.
Requires	TCPIsGetReady = TRUE

Example	`...`
	`' Fetch data`
	`if (TCPIsGetReady(httpSocket) <> 0) then`
	`' Fetch 20 bytes of data`
	`TCPGetArray(httpSocket, buffer, 20)`
	`...`

TCPDiscard

Prototype	`sub function TCPDiscard(dim socket as TCP_SOCKET) as byte`
Returns	(<>0) - If receive buffer for given socket was successfully discarded. (=0) - If receive buffer for given socket was already discarded.
Description	This function releases the receive buffer associated with a given socket. `socket` - Socket identifier that needs to be transmitted.
Requires	Nothing.

Example	`...`
	`' Fetch 20 bytes of data`
	`TCPGetArray(httpSocket, buffer, 20)`
	`' Process data.`
	`...`
	`' Release the buffer.`
	`TCPDiscard(httpSocket)`
	`...`

TCPProcess

Prototype	`sub function TCPProcess(dim byref remote as NODE_INFO, dim len as word) as byte`
Returns	(<>0) - If this function (task) has completely processed current packet. (=0) - If this function (task) has partially processed current packet.
Description	This function acts as "TCPTask". It fetches an already received TCP packet and executes the TCP State machine for matching sockets. This function must be called only when a TCP packet is received.
Requires	IPGetHeader = TRUE IPProtocol = IP_PRO_TCP

Example	`...`
	`if (TCPProcess(RemoteNode, IPDataCount) <> 0) then`
	`smState = SM_STACK_IDLE`
	`...`

TCPTick

Prototype	`sub procedure TCPTick()`
Description	This function acts as another "TCPTask" in addition to TCPProcess. This function checks time-out conditions for all sockets and attempts to recover from them.
Requires	IPGetHeader = TRUE IPProtocol = IP_PRO_TCP
Example	`TCPTick()`

UDPInit

Prototype	`sub procedure UDPInit()`
Description	This function initializes the UDP module and prepares it for multiple UDP connections.
Requires	Nothing.
Example	`' Initialize UDP` `UDPInit()`

UDPOpen

Prototype	`sub function UDPOpen(dim localPort as UDP_PORT, dim byref remoteNode as NODE_INFO, dim remotePort as TCP_PORT) as UDP_SOCKET`
Returns	A valid socket identifier if there is at least one free socket. INVALID_UDP_SOCKET if there is no socket available.
Description	This function prepares the next available UDP socket on a given port for possible data transfer. Either the local or remote node may initiate the data transfer. `localPort` - Local UDP port number on which data transfer will occur. `remoteNode` - Remote host that contains remotePort. `remotePort` - UDP Port number on remote host to transfer the data to and from.
Requires	Nothing.

Example	`...`
	`' Talk to a remote DHCP server.`
	`DHCPSocket = UDPOpen(68, DHCPServerNode, 67)`
	`if (DHCPSocket = INVALID_UDP_SOCKET) then`
	`  ' Socket is not available`
	`  ' Return error.`
	`else`
	`  ' Broadcast DHCP Broadcast message.`
	`  ...`

UDPClose

Prototype	`sub procedure UDPClose(dim socket as UDP_SOCKET)`
Description	This function closes a given UDP socket and declares it as a free socket. `socket` - Identifier of socket that needs to be closed.
Requires	Nothing.

Example	`...`
	`' Talk to a remote DHCP server.`
	`DHCPSocket = UDPOpen(68, DHCPServerNode, 67)`
	`if (DHCPSocket = INVALID_UDP_SOCKET) then`
	`  ' Socket is not available`
	`  ' Return error.`
	`else`
	`  ' Send DHCP request...`
	`  ...`
	`  ' Close the socket`
	`  UDPClose(DHCPSocket)`
	`  ...`

UDPIsPutReady

Prototype	`sub function UDPIsPutReady(dim socket as UDP_SOCKET) as byte`
Returns	(<>0) - If a given socket is ready to transmit. (=0) - If there is no transmit buffer ready.
Description	This macro determines if a given socket is ready to transmit. A socket is ready to transmit when at least one of the MAC transmit buffers is empty. It also sets the given socket as an active UDP socket. `socket` - Identifier of the socket that needs to be checked and set active.
Requires	Nothing.

Example	`' Talk to a remote DHCP server.`
	`DHCPSocket = UDPOpen(68, DHCPServerNode, 67)`
	`if (DHCPSocket = INVALID_UDP_SOCKET) then`
	`  ' Socket is not available`
	`  ' Return error.`
	`else`
	`  ' Broadcast DHCP Broadcast message.`
	`  smState = SM_BROADCAST`
	`end if`
	
	`if (UDPIsPutReady(DHCPSocket) <> 0) then`
	`  ' Socket is ready to transmit. Transmit the data...`
	`  ...`

UDPPut

Prototype	`sub function UDPPut(dim data as byte) as byte`
Returns	(<>0) - If a given data byte was successfully loaded into the transmit buffer and there is room for more data. (=0) - If a given data byte was successfully loaded into the transmit buffer and there is no room for more data.
Description	This function loads a data byte into the transmit buffer for an active socket. `data` - Data byte to be loaded.
Requires	UDPIsPutReady = TRUE

Example	`...`
	`if (UDPIsPutReady(DHCPSocket) <> 0) then`
	`  ' Socket is ready to transmit. Transmit the data...`
	`  ' Note that there is DHCPSocket parameter in UDPPut.`
	`  ' This UDPPut call will use active socket`
	`  ' as set by UDPIsPutReady() - that is DHCPSocket.`
	`  UDPPut($55)`
	`  ...`

UDPFlush

Prototype	`sub procedure UDPFlush()`
Returns	Nothing.
Description	This function marks the active socket transmit buffer as ready to be transmitted.
Requires	UDPPut() is already called and the requested UDP socket is set as the active socket by calling UDPIsPutReady().

Example	`...`
	`if (UDPIsPutReady(DHCPSocket) <> 0) then`
	`  ' Socket is ready to transmit. Transmit the data...`
	`  ' Note that there is DHCPSocket parameter in UDPPut.`
	`  ' This UDPPut call will use active socket`
	`  ' as set by UDPIsPutReady() - that is DHCPSocket.`
	`  UDPPut($55)`
	`  ...`
	`  ' Now transmit it.`
	`  UDPFlush()`
	`  ...`

UDPIsGetReady

Prototype	`sub function UDPIsGetReady(dim socket as UDP_SOCKET) as byte`
Returns	(<>0) - If a given socket contains received data. (=0) - If a given socket does not contain any data.
Description	This function determines if the given socket contains received data. It also sets a given socket as an active socket. socket - Identifier for the socket that needs to be transmitted and set active.
Requires	UDPOpen() is already called. The value of socket must be that returned by UDPOpen() call.

Example	`if (UDPIsGetReady(DHCPSocket) <> 0) then`
	`' Socket does contain some data. Fetch it and process it.`
	`...`
	`end if`
	`...`

UDPGet

Prototype	`sub function UDPGet(dim byref data as byte) as byte`
Returns	(<>0) If a byte is read. (=0) If no byte is read.
Description	This function fetches one data byte from an active socket receive buffer. `data` - Data byte that is read.
Requires	UDPIsGetReady = TRUE

Example	`...`
	`if (UDPIsGetReady(DHCPSocket) <> 0) then`
	`' Socket does contain some data. Fetch it all.`
	`' buffer is a pointer to BYTE.`
	`while (UDPGet(Buffer)<>0)`
	`  inc(buffer)`
	`  ' Process it.`
	`  ...`
	`  ' Discard the socket buffer.`
	`  ...`
	`wend`
	`...`

UDPDiscard

Prototype	`sub function UDPDiscard as byte`
Returns	Nothing.
Description	This function releases the receive buffer associated with an active socket.
Requires	Nothing.

Example	`...`
	`if (UDPIsGetReady(DHCPSocket) <> 0) then`
	`' Socket does contain some data. Fetch it all.`
	`' buffer is a pointer to BYTE.`
	`while(UDPGet(Buffer)<>0)`
	`  inc(buffer)`
	`  ' Process it.`
	`  ...`
	`  ' Discard the socket buffer.`
	`  UDPDiscard()`
	`wend`
	`...`

UDPProcess

Prototype	`sub function UDPProcess(dim byref remote as NODE_INFO, dim len as word) as byte`
Returns	(<>0) - If this function (task) has completely processed the current packet. (=0) - If this function (task) has partially processed the current packet.
Description	This function acts as "UDPTask". It fetches an already received UDP packet and assigns it to a matching UDP socket. This function must be called only when a UDP packet is received. `remote` - Remote node from which the current UDP packet was received. `len` - Total length of UDP packet length, including UDP header.
Requires	IPGetHeader = TRUE IPProtocol = IP_PRO_UDP

Example	`...`
	`if (UDPProcess(RemoteNode, IPDataCount) <> 0) then`
	`  smState = SM_STACK_IDLE`
	`  ...`

UDPWrite

Prototype	`sub function UDPWrite(dim UDPSocket as word, dim byref UDPBuffer as byte, dim NoBytes as byte) as byte`
Returns	Number of written bytes.
Description	Sends data over UDP protocol. `UDPSocket` - Writes data to this socket. `UDPBuffer` - Data to be sent. `NoBytes`.
Requires	Requires UDPInit and UDPOpenSocket.

Example	`socket = UDPOpenSocket(10001, 192,168,20,1, 10001)`
	`' assign socket for UDP protocol`
	`numofbytes = UDPWrite(socket, mydata)`

UDPRead

Prototype	`sub function UDPRead(dim UDPSocket as word, dim byref UDPBuffer as byte) as byte`
Returns	Number of read bytes.
Description	Reads data over UDP protocol. `UDPSocket` - Reads data from this socket. `UDPBuffer` - Data to be read.
Requires	Requires UDPInit and UDPOpenSocket.

Example	`socket = UDPOpenSocket(10001, 192,168,20,1, 10001)`
	`' assign socket for UDP protocol`
	`numofbytes = UDPRead(socket, mydata)`

UDPOpenSocket

Prototype	`sub function UDPOpenSocket(dim LocUDPPort as word, dim RemoteIPByte1, RemoteIPByte2, RemoteIPByte3, RemoteIPByte4 as byte, dim RemUDPPort as word) as word`
Returns	Number of initialized UDP socket.
Description	Opens socket for UDP communication with remote node. `LocUDPPort` - NIC Udp port, `RemoteIPByte1 ... RemoteIPByte4` - Remote host IP address, `RemUDPPort` - Remote host UDP port.
Requires	Requires UDPInit.
Example	`socket = UDPOpenSocket(10001, 192,168,20,1, 10001)` `' assign socket for UDP protocol`

StackInit

Prototype	`sub procedure StackInit(dim byref RstPort as word, dim RstPin as word, dim byref CSPort as word, dim CSPin as word)`
Returns	Nothing.
Description	This routine initializes stack and its components.
Requires	Nothing.
Example	`StackInit(PORTF, 0, PORTF, 1) ' initialize stack`

StackTask

Prototype	`sub procedure StackTask()`
Returns	Nothing.
Description	This routine executes Stack FSM.
Requires	StackInit() must be called before calling this routine.
Example	`StackTask()`

HTTPInit

Prototype	`sub procedure HTTPInit()`
Returns	Nothing.
Description	This routine initializes HTTP protocol.
Requires	Nothing.
Example	`HTTPInit()`

HTTPServer

Prototype	`sub procedure HTTPServer()`
Returns	Nothing.
Description	This routine starts HTTP server.
Requires	HTTPInit() must be called before using this routine.
Example	`HTTPServer()`

Library Example

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

```
program SerialEthernet

include "__Lib_SerEthIP"
include "__Lib_SerEthTCP"
include "MikroEDemoPage"


const TICKS_PER_SECOND = 100
const TICK_SECOND as longint = TICKS_PER_SECOND

const  DISPLAY          = 0
       LED_D2            = 0x00
       LED_D3            = 0x01
       ANALOGINPUT_0    = 0x02
       ANALOGINPUT_1    = 0x03
       SETLED_D2         = 0x04
       SETLED_D3         = 0x05


const HTTP_DEFAULT_PAGE as string[ 19]  = "CENTER.HTM"
const EXECUTE_PAGE as string[ 19]        = "EXECUTE.CGI"
const UNKNOWN_PAGE as string[ 20]        = "UNKNOWN.HTM"

const BaudRate        = 9600
      DefaultUDPPort = 10001

structure xxx
  dim ss as ^byte
end structure

dim main_socket as word
    Buffer0      as byte[ 10]
    Buffer1      as byte[ 10]
    Bbuffer      as byte[ 20]
    i             as word
    t             as longint

    IFS0bits     as word absolute $084 volatile register
    IEC0bits     as word absolute $08C volatile register
    T1CONbits    as word absolute $104 volatile register

// continues...
```

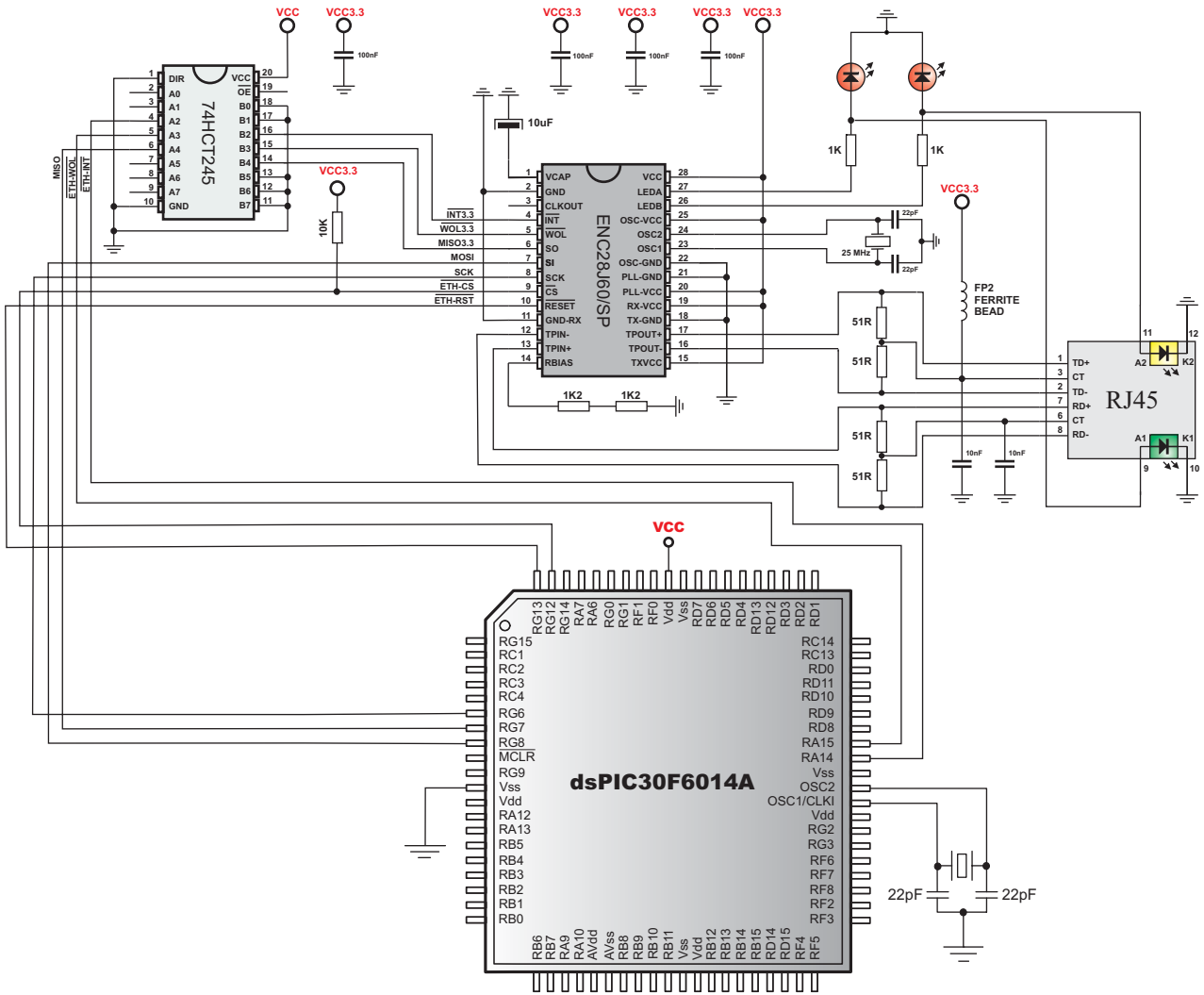
```
// continued...

sub function TickGetDiff(dim a, b as longint) as longint
    if( a < b ) then
      result = longint(0xFFFFFFFF - b) + a
    else
      result = longint( a - b )
    end if
  end sub

sub procedure Interrupt_T1() org 0x001A
    TimeOutUpdate()
  end sub

sub procedure ReadAnalogInputs()
  dim  ADCResult as word
       ADCFVal    as longint

    ADCResult = Adc_Read(0)
    ADCFVal = (ADCResult * 5000) >> 12
    WordToStrWithZeros(adcfval, Buffer0)
    Buffer0[ 5]  = Buffer0[ 4]
    Buffer0[ 4]  = Buffer0[ 3]
    Buffer0[ 3]  = Buffer0[ 2]
    Buffer0[ 0]  = " "
    Buffer0[ 2]  = "."
    Buffer0[ 6]  = " "
    Buffer0[ 7]  = "V"
    Buffer0[ 8]  = 0

    'IntToStr(ADCResult, Buffer0);
    ADCResult = Adc_Read(1)
    ADCFVal   = (ADCResult * 5000) >> 12
    WordToStrWithZeros(adcfval, Buffer1)
    Buffer1[ 5]  = Buffer1[ 4]
    Buffer1[ 4]  = Buffer1[ 3]
    Buffer1[ 3]  = Buffer1[ 2]
    Buffer1[ 0]  = " "
    Buffer1[ 2]  = "."
    Buffer1[ 6]  = " "
    Buffer1[ 7]  = "V"
    Buffer1[ 8]  = 0
  end sub


// continues...
```

```
// continued...

sub procedure HTTPExecCmd(dim byref argv as xxx[20], dim argc as
byte)
  dim  comm as byte
       i, Vvar as byte
       chptr as ^byte

    i = 0
    comm = argv[i].ss^ - "0"
    select case comm
      case DISPLAY
                  Vvar = argv[1].ss^ - "0"
                  select case Vvar
                    case LED_D2
                                LATD.2 = LATD.2 xor 1

                    case LED_D3
                                LATD.3 = LATD.3 xor 1
                  end select

                  chptr = argv[0].ss
                  for i = 0 to 12
                    chptr^ = EXECUTE_PAGE[i]
                    chptr  = chptr + 1
                  next i
      case else
                chptr = argv[0].ss
                for i = 0 to strlen(EXECUTE_PAGE)
                  chptr^ = EXECUTE_PAGE[i]
                  chptr  = chptr + 1
              next i
    end select
end sub

sub function HTTPGetVar(dim Vvar as byte, dim ref as word, dim Vval
as ^byte) as word
    select case Vvar
      case LED_D2
                        if( LATD and 0x04) <> 0 then
                          Vval^ = "1"
                        else
                          Vval^ = "0"
                        end if

// continues...
```

```
// continued...

        case LED_D3
                                if( LATD and 0x08) <> 0 then
                                  Vval^ = "1"
                                else
                                  Vval^ = "0"
                                end if

        case ANALOGINPUT_0
                                Vval^ = Buffer0[ byte(ref)]
                                if( Buffer0[ byte(ref)]  = 0 ) then
                                     result = 0xFFFF
                                     exit
                                end if
                                ref    = ref + 1
                                result = ref
                                exit

        case ANALOGINPUT_1
                                Vval^ = Buffer1[ byte(ref)]
                                if( Buffer1[ byte(ref)]  =  0 ) then
                                     result = 0xFFFF
                                     exit
                                end if
                                ref    = ref + 1
                                result = ref
                                exit

        case SETLED_D2
                                if( LATD and 0x04 ) <> 0 then
                                  Vval^ = "0"
                                else
                                  Vval^ = "1"
                                end if

        case SETLED_D3
                                if( LATD and 0x08 ) <> 0 then
                                  Vval^ = "0"
                                else
                                  Vval^ = "1"
                                end if
    end select
    result = 0xFFFF
end sub


// continues...
```

```
// continues...
main:
    i = 0
    t = 0
    ADPCFG = 0xFFFF
    PORTD = 0
    TRISD = 0
    PORTB = 0
    TRISB = 0
    ' Set up analog inputs
    TRISB = TRISB or 0x03
    ' Set up the LED's.
    LATD  = 0x00
    LATD  = LATD or 0x0C
    TRISD = 0x00

    EthSetIPAddr(192, 168, 20, 60)
    EthSetMACAddr(0x00, 0x02, 0x04, 0x06, 0x08, 0x0A)
    EthSetIPMask(0xFF, 0xFF, 0xFF, 0x00)
    EthSetGateWayAddr(192, 168, 20, 60)

    EthInit( @PORTG, 13, @PORTG, 12 )

    ' Open UDP communication sockets needed
    main_socket = UDPOpenSocket(DefaultUDPPort, 192, 168, 20, 1,
DefaultUDPPort)

    ' main loop
    while true
      if( TickGetDiff(TickGet(), t) >= TICK_SECOND/2 ) then
          t = TickGet()
          LATD = LATD xor 1
      end if

      StackTask()

        '*
        '* Process incoming UDP packets and send a reply if needed.
        '* User specific code.
        '*
      i = UDPRead(main_socket, @Bbuffer)
      if( i <> 0 ) then
          Bbuffer[ i] = 0
          UDPWrite(main_socket, @Bbuffer, strlen(Bbuffer))
      end if

      HTTPServer()
      ReadAnalogInputs()
    wend
end.
```

Hardware Connection



Button Library

The mikroBasic for dsPIC30/33 and PIC24 Button Library provides routines for detecting button presses and debouncing (eliminating the influence of contact flickering upon pressing a button).

Button

Prototype	`sub function Button(dim byref port as word, dim pin as byte, dim time as word, dim ActiveState as byte) as word`
Returns	- `1` if the pin was in the active state for given period. - `0` otherwise
Description	The function eliminates the influence of contact flickering upon pressing a button (debouncing). The Button pin is tested just after the function call and once again after the debouncing period has expired. If the pin was in the active state in both cases then the function returns 1. Parameters : - `port`: button port address - `pin`: button pin - `time`: debouncing period in milliseconds - `ActiveState`: determines what is considered as active state. Valid values: 0 (logical zero) and 1 (logical one) **Note**: There is no need to declare button pin as input. Button function does this automatically.

Example	Example reads buttons RB1..RB0 and displays button states on PORTF :
	`program Button_Test`
	`main:`
	`  ADPCFG = $FFFF`
	`  TRISF = $0000`
	`  LATF = $AAAA`
	`  Delay_ms(200)`
	`  LATF = $0000`
	`  while true`
	`  ' if button on portb.0 is 5v for 100ms`
	`  if Button(PORTB, 0, 100, 1) = 1 then`
	`  LATF = $0001`
	`  else`
	`  if Button(PORTB, 1, 100, 1) = 1 then`
	`  LATF = $0002`
	`  end if`
	`  end if`
	`  wend`

CAN Library

mikroBasic for dsPIC30/33 and PIC24 provides a library (driver) for working with the dsPIC30F CAN module.

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

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

CAN supports two message formats:

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

Note: Consult the CAN standard about CAN bus termination resistance.

Library Routines

```
CAN1SetOperationMode
CAN1GetOperationMode
CAN1Initialize
CAN1SetBaudRate
CAN1SetMask
CAN1SetFilter
CAN1Read
CAN1Write

CAN2SetOperationMode
CAN2GetOperationMode
CAN2Initialize
CAN2SetBaudRate
CAN2SetMask
CAN2SetFilter
CAN2Read
CAN2Write
```

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

```
- RegsToCAN1ID            - RegsToCAN2ID
- CAN1IDToRegs            - CAN2IDToRegs
```

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

CAN1SetOperationMode

Prototype	`sub procedure CAN1SetOperationMode(dim mode, WAIT as word)`
Returns	Nothing.
Description	Sets the CAN1 module to requested mode. Parameters : - `mode`: CAN module operation mode. Valid values: CAN_OP_MODE constants. See CAN constants. - `WAIT`: CAN1 mode switching verification request. If `WAIT = 0`, the call is non-blocking. The function does not verify if the CAN1 module is switched to requested mode or not. The caller must use CAN1GetOperationMode to verify correct operation mode before performing mode specific operation. If `WAIT <> 0`, the call is blocking – the function won't "return" until the requested mode is set.
Requires	The CAN1 routines are supported only by MCUs with the CAN1 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus.

Example	`' set the CAN1 module into configuration mode`
	`' (wait inside CAN1SetOperationMode until this mode is set)`
	`CAN1SetOperationMode(CAN_MODE_CONFIG, 0xFF)`

CAN1GetOperationMode

Prototype	`sub function CAN1GetOperationMode() as word`
Returns	Current operation mode.
Description	The function returns current operation mode of the CAN1 module. Check the `CAN_OP_MODE` constants (see CAN constants) or device datasheet for operation mode codes.
Requires	The CAN1 routines are supported only by MCUs with the CAN1 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus.

Example	`' check whether the CAN1 module is in Normal mode`
	`' and if it is then do something.`
	`if (CAN1GetOperationMode() = CAN_MODE_NORMAL) then`
	`  ...`
	`end if`

CAN1Initialize

Prototype	`sub procedure CAN1Initialize(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS as word)`
Description	Initializes the CAN1 module. The internal dsPIC30F CAN1 module is set to: - Disable CAN capture - Continue CAN operation in Idle mode - Do not abort pending transmissions - Fcan clock : 4*Tcy (Fosc) - Baud rate is set according to the given parameters - CAN mode is set to Normal - Filter and mask registers IDs are set to zero - Filter and mask message frame type is set according to the `CAN_CONFIG_FLAGS` value The `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to the `CAN_CONFIG_FLAGS` value. Parameters: - `SJW` as defined in MCU's datasheet (CAN1 Module) - `BRP` as defined in MCU's datasheet (CAN1 Module) - `PHSEG1` as defined in MCU's datasheet (CAN1 Module) - `PHSEG2` as defined in MCU's datasheet (CAN1 Module) - `PROPSEG` as defined in MCU's datasheet (CAN1 Module) - `CAN_CONFIG_FLAGS` is formed from predefined constants. See CAN constants **Note**: CAN mode NORMAL will be set on exit.
Requires	The CAN1 routines are supported only by MCUs with the CAN1 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus.

Example	`' initialize the CAN1 module with appropriate baud rate and`
	`' message acceptance flags along with the sampling rules`
	`dim can_config_flags as word`
	`...`
	`can_config_flags = CAN_CONFIG_SAMPLE_THRICE and`
	`CAN_CONFIG_PHSEG2_PRG_ON and`
	`CAN_CONFIG_STD_MSG and`
	`CAN_CONFIG_DBL_BUFFER_ON and`
	`CAN_CONFIG_MATCH_MSG_TYPE and`
	`CAN_CONFIG_LINE_FILTER_OFF`
	
	`CAN1Initialize(1,3,3,3,1,can_config_flags)`
	`' initialize the CAN1 module`

CAN1SetBaudRate

Prototype	`sub procedure CAN1SetBaudRate(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS as word)`
Description	Sets CAN1 baud rate. Due to complexity of the CAN protocol, you can not simply force a bps value. Instead, use this function when CAN1 is in Config mode. Refer to datasheet for details. The `SAM`, `SEG2PHTS` and `WAKFIL` bits are set according to the `CAN_CONFIG_FLAGS` value. Refer to datasheet for details. Parameters: - `SJW` as defined in MCU's datasheet (CAN1 Module) - `BRP` as defined in MCU's datasheet (CAN1 Module) - `PHSEG1` as defined in MCU's datasheet (CAN1 Module) - `PHSEG2` as defined in MCU's datasheet (CAN1 Module) - `PROPSEG` as defined in MCU's datasheet (CAN1 Module) - `CAN_CONFIG_FLAGS` is formed from predefined constants. See CAN constants
Requires	The CAN1 routines are supported only by MCUs with the CAN1 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus. CAN1 must be in Config mode, otherwise the function will be ignored. See CAN1SetOperationMode.

Example	`' set required baud rate and sampling rules`
	`dim can_config_flags as word`
	`...`
	`CAN1SetOperationMode(CAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CAN1 module must be in config mode`
	`' for baud rate settings)`
	
	`can_config_flags = CAN_CONFIG_SAMPLE_THRICE and`
	`' Form value to be used`
	`CAN_CONFIG_PHSEG2_PRG_ON and`
	`' with CAN1Initialize`
	`CAN_CONFIG_STD_MSG and`
	`CAN_CONFIG_DBL_BUFFER_ON and`
	`CAN_CONFIG_MATCH_MSG_TYPE and`
	`CAN_CONFIG_LINE_FILTER_OFF`
	
	`CAN1SetBaudRate(1,3,3,3,1,can_config_flags)`
	`' set the CAN1 module baud rate`

CAN1SetMask

Prototype	`sub procedure CAN1SetMask(dim CAN_MASK as word, dim val as longint, dim CAN_CONFIG_FLAGS as word)`
Description	The function sets mask for advanced filtering of messages. Parameters: - `CAN_MASK`: CAN module mask number. Valid values: `CAN_MASK` constants. See CAN constants - `val`: mask register value. This value is bit-adjusted to appropriate buffer mask registers - `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values: `CAN_CONFIG_ALL_VALID_MSG`, `CAN_CONFIG_MATCH_MSG_TYPE` and `CAN_CONFIG_STD_MSG`, `CAN_CONFIG_MATCH_MSG_TYPE` and `CAN_CONFIG_XTD_MSG`. (see CAN constants)
Requires	The CAN1 routines are supported only by MCUs with the CAN1 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus. CAN1 must be in Config mode, otherwise the function will be ignored. See CAN1SetOperationMode.

Example	`' set appropriate filter mask and message type value`
	`CAN1SetOperationMode(CAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CAN1 module must be in config mode`
	`' for mask settings)`
	
	`' Set all B1 mask bits to 1 (all filtered bits are relevant)`
	`' Note that -1 is just a cheaper way to write 0xFFFFFFFF.`
	`' Complement will do the trick and fill it up with ones.`
	`CAN1SetMask(CAN_MASK_B1, -1, CAN_CONFIG_MATCH_MSG_TYPE and CAN_CONFIG_XTD_MSG)`

CAN1SetFilter

Prototype	`sub procedure CAN1SetFilter(dim CAN_FILTER as word, dim val as longint, dim CAN_CONFIG_FLAGS as word)`
Description	The function sets message filter. Parameters: - `CAN_FILTER`: CAN module filter number. Valid values: `CAN_FILTER` constants. See CAN constants - `val`: filter register value. This value is bit-adjusted to appropriate filter registers - `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values: `CAN_CONFIG_STD_MSG` and `CAN_CONFIG_XTD_MSG`. See CAN constants
Requires	The CAN1 routines are supported only by MCUs with the CAN1 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus. CAN1 must be in Config mode, otherwise the function will be ignored. See CAN1SetOperationMode.

Example	`' set appropriate filter value and message type`
	`CAN1SetOperationMode(CAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CAN1 module must be`
	`' in config mode for filter settings)`
	
	`' Set id of filter B1_F1 to 3:`
	`CAN1SetFilter(CAN_FILTER_B1_F1, 3, CAN_CONFIG_XTD_MSG)`

CAN1Read

Prototype	`sub function CAN1Read(dim byref id as longint, dim byref data as byte[1], dim dataLen, CAN_RX_MSG_FLAGS as word) as word`
Returns	- `0` if nothing is received - `0xFFFF` if one of the Receive Buffers is full (message received)
Description	If at least one full Receive Buffer is found, it will be processed in the following way: - Message ID is retrieved and stored to location pointed by id pointer - Message data is retrieved and stored to array pointed by data pointer - Message length is retrieved and stored to location pointed by dataLen pointer - Message flags are retrieved and stored to location pointed by `CAN_RX_MSG_FLAGS` pointer Parameters: - `id`: message identifier address - `data`: an array of bytes up to 8 bytes in length - `dataLen`: data length address - `CAN_RX_MSG_FLAGS`: message flags address. For message receive flags format refer to the `CAN_RX_MSG_FLAGS` constants (see CAN constants).
Requires	The CAN1 routines are supported only by MCUs with the CAN1 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus. The CAN1 module must be in a mode in which receiving is possible. See CAN1SetOperationMode.

Example	`' check the CAN1 module for received messages.`
	`' If any was received do something.`
	`dim msg_rcvd, rx_flags, data_len as word`
	`  data as byte[8]`
	`  msg_id as longint`
	`...`
	`CAN1SetOperationMode(CAN_MODE_NORMAL,0xFF)  ' set NORMAL mode`
	`'(CAN1 module must be in mode in which receive is possible)`
	`...`
	`rx_flags = 0  ' clear message flags`
	`if (msg_rcvd = CAN1Read(msg_id, data, data_len, rx_flags) <> 0) then`
	`  ...`
	`end if`

CAN1Write

Prototype	`sub function CAN1Write(dim id as longint, dim byref data as byte[1], dim dataLen, CAN_TX_MSG_FLAGS as word) as word`
Returns	- `0` if all Transmit Buffers are busy - `0xFFFF` if at least one Transmit Buffer is available
Description	If at least one empty Transmit Buffer is found, the function sends message in the queue for transmission. Parameters: - `id`: CAN1 message identifier. Valid values: 11 or 29 bit values, depending on message type (standard or extended) - `Data`: data to be sent - `DataLen`: data length. Valid values: 0..8 - `CAN_TX_MSG_FLAGS`: message flags. Valid values: `CAN_TX_MSG_FLAGS` constants. See CAN constants.
Requires	The CAN1 routines are supported only by MCUs with the CAN1 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus. The CAN1 module must be in mode in which transmission is possible. See CAN1SetOperationMode.

Example	`' send message extended CAN message with appropriate ID and data`
	`dim tx_flags as word`
	`  data as byte[8]`
	`  msg_id as longint`
	`...`
	`CAN1SetOperationMode(CAN_MODE_NORMAL,0xFF)  ' set NORMAL mode`
	`' (CAN1 must be in mode in which transmission is possible)`
	
	`tx_flags = CAN_TX_PRIORITY_0 and`
	`  CAN_TX_XTD_FRAME and`
	`  CAN_TX_NO_RTR_FRAME`
	`' set message flags`
	`CAN1Write(msg_id, data, 1, tx_flags)`

CAN2SetOperationMode

Prototype	`sub procedure CAN2SetOperationMode(dim mode, WAIT as word)`
Description	Sets the CAN2 module to requested mode. Parameters : - `mode`: CAN2 module operation mode. Valid values: CAN_OP_MODE constants. See CAN constants. - `WAIT`: CAN2 mode switching verification request. If `WAIT = 0`, the call is non-blocking. The function does not verify if the CAN2 module is switched to requested mode or not. The caller must use CAN2GetOperationMode to verify correct operation mode before performing mode specific operation. If `WAIT <> 0`, the call is blocking – the function won't "return" until the requested mode is set.
Requires	The CAN2 routines are supported only by MCUs with the CAN2 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus.

Example	`' set the CAN2 module into configuration mode`
	`' (wait inside CAN2SetOperationMode until this mode is set)`
	`CAN2SetOperationMode(CAN_MODE_CONFIG, 0xFF)`

CAN2GetOperationMode

Prototype	`sub function CAN2GetOperationMode() as word`
Returns	Current operation mode.
Description	The function returns current operation mode of the CAN2 module. Check `CAN_OP_MODE` constants (see CAN constants) or device datasheet for operation mode codes.
Requires	The CAN2 routines are supported only by MCUs with the CAN2 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus.

Example	`' check whether the CAN2 module is in Normal mode`
	`' and if it is then do something.`
	`if (CAN2GetOperationMode() = CAN_MODE_NORMAL) then`
	
	`...`
	`end if`

CAN2Initialize

Prototype	`sub procedure CAN2Initialize(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS as word)`
Description	Initializes the CAN2 module. The internal dsPIC30F CAN2 module is set to: - Disable CAN capture - Continue CAN operation in Idle mode - Do not abort pending transmissions - Fcan clock : 4*Tcy (Fosc) - Baud rate is set according to the given parameters - CAN mode is set to Normal - Filter and mask registers IDs are set to zero - Filter and mask message frame type is set according to the `CAN_CONFIG_FLAGS` value The `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to the `CAN_CONFIG_FLAGS` value. Parameters: - `SJW` as defined in MCU's datasheet (CAN2 Module) - `BRP` as defined in MCU's datasheet (CAN2 Module) - `PHSEG1` as defined in MCU's datasheet (CAN2 Module) - `PHSEG2` as defined in MCU's datasheet (CAN2 Module) - `PROPSEG` as defined in MCU's datasheet (CAN2 Module) - `CAN_CONFIG_FLAGS` is formed from predefined constants. See CAN constants **Note**: CAN mode NORMAL will be set on exit.
Requires	The CAN2 routines are supported only by MCUs with the CAN2 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus.

Example	`' initialize the CAN2 module with appropriate baud rate`
	`' and message acceptance flags along with the sampling rules`
	`dim can_config_flags as word`
	`...`
	`can_config_flags = CAN_CONFIG_SAMPLE_THRICE and`
	`' Form value to be used`
	`CAN_CONFIG_PHSEG2_PRG_ON and`
	`' with CAN2Initialize`
	`CAN_CONFIG_STD_MSG and`
	`CAN_CONFIG_DBL_BUFFER_ON and`
	`CAN_CONFIG_MATCH_MSG_TYPE and`
	`CAN_CONFIG_LINE_FILTER_OFF`
	
	`CAN2Initialize(1,3,3,3,1,can_config_flags) ' initialize the CAN2`

CAN2SetBaudRate

Prototype	`sub procedure CAN2SetBaudRate(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS as word)`
Description	Sets CAN2 baud rate. Due to complexity of the CAN protocol, you can not simply force a bps value. Instead, use this function when CAN2 is in Config mode. The `SAM`, `SEG2PHTS` and `WAKFIL` bits are set according to the `CAN_CONFIG_FLAGS` value. Refer to datasheet for details. Parameters: - `SJW` as defined in MCU's datasheet (CAN2 Module) - `BRP` as defined in MCU's datasheet (CAN2 Module) - `PHSEG1` as defined in MCU's datasheet (CAN2 Module) - `PHSEG2` as defined in MCU's datasheet (CAN2 Module) - `PROPSEG` as defined in MCU's datasheet (CAN2 Module) - `CAN_CONFIG_FLAGS` is formed from predefined constants. See CAN constants.
Requires	The CAN2 routines are supported only by MCUs with the CAN2 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus. CAN2 must be in Config mode, otherwise the function will be ignored. See CAN2SetOperationMode.

Example	`' set required baud rate and sampling rules`
	`dim can_config_flags as word`
	`...`
	`CAN2SetOperationMode(CAN_MODE_CONFIG,0xFF)' set CONFIGURATION mode`
	`' (CAN2 module must be in config mode for baud rate settings)`
	
	`can_config_flags = CAN_CONFIG_SAMPLE_THRICE and`
	`' Form value to be used`
	`CAN_CONFIG_PHSEG2_PRG_ON and`
	`' with CAN2SetBaudRate`
	`CAN_CONFIG_STD_MSG and`
	`CAN_CONFIG_DBL_BUFFER_ON and`
	`CAN_CONFIG_MATCH_MSG_TYPE and`
	`CAN_CONFIG_LINE_FILTER_OFF`
	
	`CAN2SetBaudRate(1,3,3,3,1,can_config_flags)`
	`' set the CAN2 module baud rate`

CAN2SetMask

Prototype	`sub procedure CAN2SetMask(dim CAN_MASK as word, dim val as longint, dim CAN_CONFIG_FLAGS as word)`
Description	The function sets mask for advanced filtering of messages. Parameters: - `CAN_MASK`: CAN2 module mask number. Valid values: CAN_MASK constants. See CAN constants - `val`:mask register value. This value is bit-adjusted to appropriate buffer mask registers - `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values: `CAN_CONFIG_ALL_VALID_MSG`, `CAN_CONFIG_MATCH_MSG_TYPE` and `CAN_CONFIG_STD_MSG`, `CAN_CONFIG_MATCH_MSG_TYPE` and `CAN_CONFIG_XTD_MSG`. (see CAN constants)
Requires	The CAN2 routines are supported only by MCUs with the CAN2 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus. CAN2 must be in Config mode, otherwise the function will be ignored. See CAN2SetOperationMode.

Example	`' set appropriate filter mask and message type value`
	`CAN2SetOperationMode(CAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CAN2 module must be in config mode`
	`' for mask settings)`
	
	`' Set all B1 mask bits to 1 (all filtered bits are relevant):`
	`' Note that -1 is just a cheaper way to write 0xFFFFFFFF.`
	`' Complement will do the trick and fill it up with ones.`
	`CAN2SetMask(CAN_MASK_B1, -1, CAN_CONFIG_MATCH_MSG_TYPE and CAN_CONFIG_XTD_MSG)`

CAN2SetFilter

Prototype	`sub procedure CAN2SetFilter(dim CAN_FILTER as word, dim val as longint, dim CAN_CONFIG_FLAGS as word)`
Description	The function sets message filter. Parameters: - `CAN_FILTER`: CAN2 module filter number. Valid values: `CAN_FILTER` constants (see CAN constants) - `val`: filter register value. This value is bit-adjusted to appropriate filter registers - `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values: `CAN_CONFIG_STD_MSG` and `CAN_CONFIG_XTD_MSG`. See CAN constants
Requires	The CAN2 routines are supported only by MCUs with the CAN2 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus. CAN2 must be in Config mode, otherwise the function will be ignored. See CAN2SetOperationMode.

Example	`' set appropriate filter value and message type`
	`CAN2SetOperationMode(CAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CAN2 module must be`
	`' in config mode for filter settings)`
	
	`' Set id of filter B1_F1 to 3:`
	`CAN2SetFilter(CAN_FILTER_B1_F1, 3, CAN_CONFIG_XTD_MSG)`

CAN2Read

Prototype	`sub function CAN2Read(dim byref id as longint, dim byref data as byte[ 1] , dim dataLen, CAN_RX_MSG_FLAGS as word) as word`
Returns	- `0` if nothing is received - `0xFFFF` if one of the Receive Buffers is full (message received)
Description	If at least one full Receive Buffer is found, it will be processed in the following way: - Message ID is retrieved and stored to location pointed by id pointer - Message data is retrieved and stored to array pointed by data pointer - Message length is retrieved and stored to location pointed by dataLen pointer - Message flags are retrieved and stored to location pointed by `CAN_RX_MSG_FLAGS` pointer Parameters: - `id`: message identifier address - `data`: an array of bytes up to 8 bytes in length - `dataLen`: data length address. - `CAN_RX_MSG_FLAGS`: message flags address. For message receive flags format refer to the `CAN_RX_MSG_FLAGS` constants (see CAN constants).
Requires	The CAN2 routines are supported only by MCUs with the CAN2 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus. The CAN2 module must be in a mode in which receiving is possible. See CAN2SetOperationMode.

Example	`' check the CAN2 module for received messages.`
	`' If any was received do something.`
	`dim msg_rcvd, rx_flags, data_len as word`
	`  data as byte[ 8]`
	`  msg_id as longint`
	`...`
	`CAN2SetOperationMode(CAN_MODE_NORMAL,0xFF)  ' set NORMAL mode`
	`' (CAN2 module must be in mode in which receive is possible)`
	`...`
	`rx_flags = 0`
	`' clear message flags`
	`if (msg_rcvd = CAN2Read(msg_id, data, data_len, rx_flags)) then`
	`  ...`
	`end if`

CAN2Write

Prototype	`sub function CAN2Write(dim id as longint, dim byref data as byte[ 1] , dim dataLen, CAN_TX_MSG_FLAGS as word) as word`
Returns	- `0` if all Transmit Buffers are busy - `0xFFFF` if at least one Transmit Buffer is available
Description	If at least one empty Transmit Buffer is found, the function sends message in the queue for transmission. Parameters: - `id`: CAN2 message identifier. Valid values: 11 or 29 bit values, depending on message type (standard or extended) - `Data`: data to be sent - `DataLen`: data length. Valid values: 0..8 - `CAN_TX_MSG_FLAGS`: message flags. Valid values: `CAN_TX_MSG_FLAGS` constants. See CAN constants
Requires	The CAN2 routines are supported only by MCUs with the CAN2 module. MCU must be connected to the CAN transceiver (MCP2551 or similar) which is connected to the CAN bus. The CAN2 module must be in mode in which transmission is possible. See CAN2SetOperationMode.

Example	`' send message extended CAN message with appropriate ID and data`
	`dim tx_flags as word`
	`  data as byte[ 8]`
	`  msg_id as longint`
	`...`
	`CAN2SetOperationMode(CAN_MODE_NORMAL,0xFF)  ' set NORMAL mode`
	`' (CAN2 must be in mode in which transmission is possible)`
	
	`tx_flags  = ECAN_TX_PRIORITY_0 and`
	`  ECAN_TX_XTD_FRAME  and`
	`  ECAN_TX_NO_RTR_FRAME`
	`' set message flags`
	`CAN2Write(msg_id, data, 1, tx_flags)`

CAN Constants

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

CAN_OP_MODE

CAN_OP_MODE constants define CAN operation mode. The functions CAN1SetOperationMode and CAN2SetOperationMode expect one of them as their argument:

```
const     CAN_MODE_BITS        as word = $E0
          CAN_MODE_NORMAL      as word = 0x00
          CAN_MODE_DISABLE     as word = 0x01
          CAN_MODE_LOOP        as word = 0x02
          CAN_MODE_LISTEN      as word = 0x03
          CAN_MODE_CONFIG      as word = 0x04
          CAN_MODE_LISTEN_ALL as word = 0x07
```

CAN_CONFIG_FLAGS

CAN_CONFIG_FLAGS constants define flags related to the CAN module configuration. The functions CAN1Initialize, CAN1SetBaudRate, CAN2Initialize, CAN2SetBaudRate expect one of these (or a bitwise combination) as their argument:

```
const
    CAN_CONFIG_DEFAULT          as word = 0xFF    ' 11111111

    CAN_CONFIG_PHSEG2_PRG_BIT  as word = 0x01
    CAN_CONFIG_PHSEG2_PRG_ON   as word = 0xFF    ' XXXXXXX1
    CAN_CONFIG_PHSEG2_PRG_OFF  as word = 0xFE    ' XXXXXXX0

    CAN_CONFIG_LINE_FILTER_BIT as word = 0x02
    CAN_CONFIG_LINE_FILTER_ON  as word = 0xFF    ' XXXXXX1X
    CAN_CONFIG_LINE_FILTER_OFF as word = 0xFD    ' XXXXXX0X

    CAN_CONFIG_SAMPLE_BIT       as word = 0x04
    CAN_CONFIG_SAMPLE_ONCE      as word = 0xFF    ' XXXXX1XX
    CAN_CONFIG_SAMPLE_THRICE    as word = 0xFB    ' XXXXX0XX

    CAN_CONFIG_MSG_TYPE_BIT     as word = 0x08
    CAN_CONFIG_STD_MSG          as word = 0xFF    ' XXXX1XXX
    CAN_CONFIG_XTD_MSG          as word = 0xF7    ' XXXX0XXX
' continues...
```

```
' ..continued

    CAN_CONFIG_DBL_BUFFER_BIT   as word = 0x10
    CAN_CONFIG_DBL_BUFFER_ON    as word = 0xFF    ' XXX1XXXX
    CAN_CONFIG_DBL_BUFFER_OFF   as word = 0xEF    ' XXX0XXXX

    CAN_CONFIG_MATCH_TYPE_BIT   as word = 0x20
    CAN_CONFIG_ALL_VALID_MSG    as word = 0xDF    ' XX0XXXXX
    CAN_CONFIG_MATCH_MSG_TYPE   as word = 0xFF    ' XX1XXXXX
```

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

```
init  = CAN_CONFIG_SAMPLE_THRICE     and
        CAN_CONFIG_PHSEG2_PRG_ON     and
        CAN_CONFIG_STD_MSG            and
        CAN_CONFIG_DBL_BUFFER_ON     and
        CAN_CONFIG_VALID_XTD_MSG     and
        CAN_CONFIG_LINE_FILTER_OFF
...
CANInitialize(1, 1, 3, 3, 1, init)    ' initialize CAN
```

CAN_TX_MSG_FLAGS

CAN_TX_MSG_FLAGS are flags related to transmission of a CAN message:

```
const
    CAN_TX_PRIORITY_BITS as word = 0x03
    CAN_TX_PRIORITY_0    as word = 0xFC    ' XXXXXX00
    CAN_TX_PRIORITY_1    as word = 0xFD    ' XXXXXX01
    CAN_TX_PRIORITY_2    as word = 0xFE    ' XXXXXX10
    CAN_TX_PRIORITY_3    as word = 0xFF    ' XXXXXX11

    CAN_TX_FRAME_BIT     as word = 0x08
    CAN_TX_STD_FRAME     as word = 0xFF    ' XXXXX1XX
    CAN_TX_XTD_FRAME     as word = 0xF7    ' XXXXX0XX

    CAN_TX_RTR_BIT       as word = 0x40
    CAN_TX_NO_RTR_FRAME  as word = 0xFF    ' X1XXXXXX
    CAN_TX_RTR_FRAME     as word = 0xBF    ' X0XXXXXX
```

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

```
' form value to be used with CANSendMessage:
send_config = CAN_TX_PRIORITY_0       and
              CAN_TX_XTD_FRAME        and
              CAN_TX_NO_RTR_FRAME
...
CANSendMessage(id, data, 1, send_config)
```

CAN_RX_MSG_FLAGS

CAN_RX_MSG_FLAGS are flags related to reception of CAN message. If a particular bit is set then corresponding meaning is TRUE otherwise it will be FALSE.

```
const
    CAN_RX_FILTER_BITS  as word = 0x0007
    CAN_RX_FILTER_1     as word = 0x00
    CAN_RX_FILTER_2     as word = 0x01
    CAN_RX_FILTER_3     as word = 0x02
    CAN_RX_FILTER_4     as word = 0x03
    CAN_RX_FILTER_5     as word = 0x04
    CAN_RX_FILTER_6     as word = 0x05
    CAN_RX_OVERFLOW     as word = 0x08
    CAN_RX_INVALID_MSG  as word = 0x10
    CAN_RX_XTD_FRAME    as word = 0x20
    CAN_RX_RTR_FRAME    as word = 0x40
    CAN_RX_DBL_BUFFERED as word = 0x80
```

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

```
if (MsgFlag and CAN_RX_OVERFLOW) <> 0 then
  ...
  ' Receiver overflow has occurred.
  ' We have lost our previous message.
end if
```

CAN_MASK

CAN_MASK constants define mask codes. The functions CAN1SetMask and CAN2SetMask expect one of these as their argument:

```
const
     CAN_MASK_B1 as word = 0
     CAN_MASK_B2 as word = 1
```

CAN_FILTER

CAN_FILTER constants define filter codes. The functions CAN1SetFilter and CAN2SetFilter expect one of these as their argument:

```
const
    CAN_FILTER_B1_F1 as word = 0
    CAN_FILTER_B1_F2 as word = 1
    CAN_FILTER_B2_F1 as word = 2
    CAN_FILTER_B2_F2 as word = 3
    CAN_FILTER_B2_F3 as word = 4
    CAN_FILTER_B2_F4 as word = 5
```

Library Example

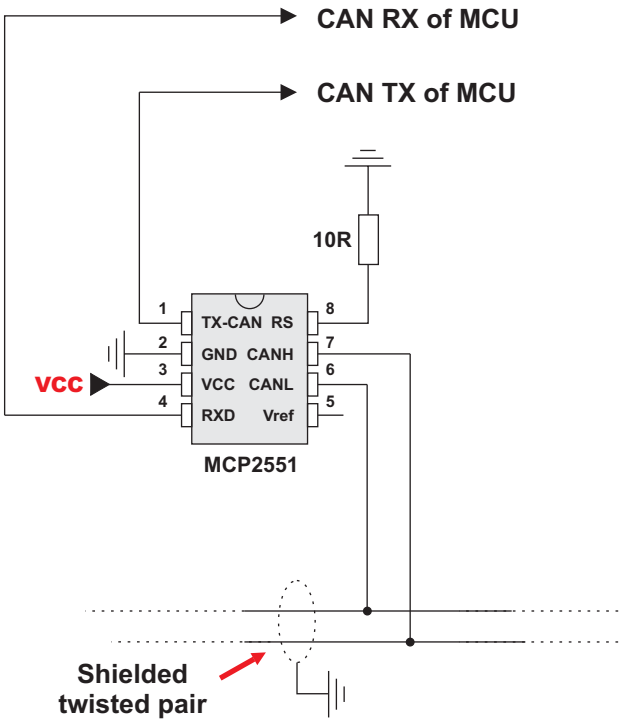
The example demonstrates CAN protocol. The 1st node initiates the communication with the 2nd node by sending some data to its address. The 2nd node responds by sending back the data incremented by 1. The 1st node then does the same and sends incremented data back to the 2nd node, etc.

```
program Can1st

dim Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags, Rx_Data_Len  as word
    RxTx_Data  as byte[ 8]
    Tx_ID, Rx_ID as longint
    Msg_Rcvd as word

main:

  ADPCFG = 0xFFFF
  PORTB  = 0
  TRISB  = 0

  Can_Init_Flags    = 0
  Can_Send_Flags    = 0
  Can_Rcv_Flags     = 0

  Can_Send_Flags    = CAN_TX_PRIORITY_0 and            ' Form value to be used
                      CAN_TX_XTD_FRAME and             '  with CANSendMessage
                      CAN_TX_NO_RTR_FRAME

  Can_Init_Flags    = CAN_CONFIG_SAMPLE_THRICE and     ' Form value to be used
                      CAN_CONFIG_PHSEG2_PRG_ON and     '  with CANInitialize
                      CAN_CONFIG_XTD_MSG and
                      CAN_CONFIG_DBL_BUFFER_ON and
                      CAN_CONFIG_MATCH_MSG_TYPE and
                      CAN_CONFIG_LINE_FILTER_OFF

  RxTx_Data[ 0]  = 9
  CAN1Initialize(1,3,3,3,1,Can_Init_Flags)            ' initialize CAN
  CAN1SetOperationMode(CAN_MODE_CONFIG,0xFF)          ' set CONFIGURATION mode

  CAN1SetMask(CAN_MASK_B1, -1, CAN_CONFIG_MATCH_MSG_TYPE and CAN_CONFIG_XTD_MSG)
  ' set all mask1 bits to ones
  CAN1SetMask(CAN_MASK_B2, -1, CAN_CONFIG_MATCH_MSG_TYPE and CAN_CONFIG_XTD_MSG)
  ' set all mask2 bits to ones
  CAN1SetFilter(CAN_FILTER_B2_F3,3,CAN_CONFIG_XTD_MSG)  'set id of filter B2_F3 to 3

  CAN1SetOperationMode(CAN_MODE_NORMAL,0xFF)            ' set NORMAL mode

'continues ...
```

```
' continued...

  Tx_ID = 12111

  CAN1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags)

  while TRUE
    Msg_Rcvd = CAN1Read(Rx_ID , RxTx_Data , Rx_Data_Len, Can_Rcv_Flags)
    if ((Rx_ID = 3) and (Msg_Rcvd <> 0)) <> 0 then
      PORTB = RxTx_Data[ 0]                                  ' output data at PORTB
      RxTx_Data[ 0]  = RxTx_Data[ 0]  + 1
      Delay_ms(10)
      CAN1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags)  ' send incremented data back
    end if
  wend
end.

program Can2nd

dim Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags, Rx_Data_Len as word
    RxTx_Data as byte[ 8]
    Tx_ID, Rx_ID as longint
    Msg_Rcvd as word
main:
  ADPCFG = 0xFFFF
  PORTB  = 0
  TRISB  = 0
  Can_Init_Flags  = 0
  Can_Send_Flags  = 0
  Can_Rcv_Flags   = 0
  Can_Send_Flags =  CAN_TX_PRIORITY_0 and              ' Form value to be used
                     CAN_TX_XTD_FRAME and              '  with CANSendMessage
                     CAN_TX_NO_RTR_FRAME
  Can_Init_Flags  =  CAN_CONFIG_SAMPLE_THRICE and      ' Form value to be used
                      CAN_CONFIG_PHSEG2_PRG_ON and      '  with CANInitialize
                      CAN_CONFIG_XTD_MSG and
                      CAN_CONFIG_DBL_BUFFER_ON and
                      CAN_CONFIG_MATCH_MSG_TYPE and
                      CAN_CONFIG_LINE_FILTER_OFF

  CAN1Initialize(1,3,3,3,1,Can_Init_Flags)             ' initialize CAN
  CAN1SetOperationMode(CAN_MODE_CONFIG,0xFF)           ' set CONFIGURATION mode

  CAN1SetMask(CAN_MASK_B1, -1, CAN_CONFIG_MATCH_MSG_TYPE and CAN_CONFIG_XTD_MSG)
  ' set all mask1 bits to ones
  CAN1SetMask(CAN_MASK_B2, -1, CAN_CONFIG_MATCH_MSG_TYPE and CAN_CONFIG_XTD_MSG)
  ' set all mask2 bits to ones
  CAN1SetFilter(CAN_FILTER_B1_F1,12111,CAN_CONFIG_XTD_MSG)
  CAN1SetOperationMode(CAN_MODE_NORMAL,0xFF)  ' set NORMAL mode
' continues...
```

```
' continued...

  Tx_ID = 3

  while TRUE
    Msg_Rcvd = CAN1Read(Rx_ID , RxTx_Data , Rx_Data_Len, Can_Rcv_Flags)
    if ((Rx_ID = 12111) and (Msg_Rcvd <> 0)) <> 0 then
      PORTB    = RxTx_Data[ 0]  ' output data at PORTB
      RxTx_Data[ 0]  = RxTx_Data[ 0]  + 1
      CAN1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags)  ' send incremented data back
    end if
  wend
end.
```

Hardware Connection



CANSPI Library

The SPI module is available with a number of the dsPIC30/33 and PIC24 MCUs. mikroBasic for dsPIC30/33 and PIC24 provides a library (driver) for working with mikroElektronika's CANSPI Add-on boards (with MCP2515 or MCP2510) via SPI interface.

In mikroBasic for dsPIC30/33 and PIC24, each routine of the CAN library has its own CANSPI counterpart with identical syntax. For more information on Controller Area Network, consult the CAN Library. Note that an effective communication speed depends on SPI and certainly is slower than "real" CAN.

Note:

- CANSPI1 library uses SPI1 module for SPI interface.
- CANSPI2 library uses SPI2 module for SPI interface.

Note:

- CANSPI1 module refers to mikroElektronika's CANSPI Add-on board connected to SPI1 module of MCU.
- CANSPI2 module refers to mikroElektronika's CANSPI Add-on board connected to SPI2 module of MCU.

Library Routines

```
CANSPI1SetOperationMode        CANSPI2SetOperationMode
CANSPI1GetOperationMode        CANSPI2GetOperationMode
CANSPI1Initialize              CANSPI2Initialize
CANSPI1SetBaudRate             CANSPI2SetBaudRate
CANSPI1SetMask                 CANSPI2SetMask
CANSPI1SetFilter               CANSPI2SetFilter
CANSPI1Read                    CANSPI2Read
CANSPI1Write                   CANSPI2Write
```

Following routines are for the internal use by compiler only:

```
RegsToCANSPI1ID          RegsToCANSPI2ID
CANSPI1IDToRegs          CANSPI2IDToRegs
```

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

CANSPI1SetOperationMode

Prototype	`sub procedure CANSPI1SetOperationMode(dim mode as byte, dim WAIT as byte)`
Description	Sets the CANSPI1 module to requested mode. Parameters : - `mode`: CANSPI1 module operation mode. Valid values: `CANSPI_OP_MODE` constants (see CANSPI constants). - `WAIT`: CANSPI1 mode switching verification request. If `WAIT = 0`, the call is non-blocking. The function does not verify if the CANSPI1 module is switched to requested mode or not. The caller must use `CANSPI1GetOperationMode` to verify correct operation mode before performing mode specific operation. If `WAIT <> 0`, the call is blocking – the function won't "return" until the requested mode is set.
Requires	The CANSPI1 routines are supported only by MCUs with the SPI1 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' set the CANSPI1 module into configuration mode (wait inside`
	`' CANSPI1SetOperationMode until this mode is set)`
	`CANSPI1SetOperationMode(CANSPI_MODE_CONFIG, 0xFF)`

CANSPI1GetOperationMode

Prototype	`sub function CANSPI1GetOperationMode() as byte`
Returns	Current operation mode.
Description	The function returns current operation mode of the CANSPI1 module. Check the `CANSPI_OP_MODE` constants (see CANSPI constants) or device datasheet for operation mode codes.
Requires	The CANSPI1 routines are supported only by MCUs with the SPI1 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' check whether the CANSPI1 module is in Normal mode`
	`' and if it is do something.`
	`if (CANSPI1GetOperationMode() = CANSPI_MODE_NORMAL) then`
	`  ...`
	`end if`

CANSPI1Init

Prototype	`sub procedure CANSPI1Init(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS as byte, dim byref RstPort as word, dim RstPin as word, dim byref CSPort as word, dim CSPin as word)`
Description	Initializes the CANSPI1 module. Stand-Alone CAN controller in the CANSPI module is set to: - Disable CAN capture - Continue CAN operation in Idle mode - Do not abort pending transmissions - Fcan clock : 4*Tcy (Fosc) - Baud rate is set according to the given parameters - CAN mode : Normal - Filter and mask registers IDs are set to zero - Filter and mask message frame type is set according to the `CAN_CONFIG_FLAGS` value `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to the `CAN_CONFIG_FLAGS` value. Parameters: - `SJW` as defined in CAN controller's datasheet - `BRP` as defined in CAN controller's datasheet - `PHSEG1` as defined in CAN controller's datasheet - `PHSEG2` as defined in CAN controller's datasheet - `PROPSEG` as defined in CAN controller's datasheet - `CAN_CONFIG_FLAGS` is formed from predefined constants (see CANSPI constants)
Requires	The CANSPI1 routines are supported only by MCUs with the SPI1 module. The SPI1 module needs to be initialized. See the Spi1_Init and Spi1_Init_Advanced routines. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`dim can_config_flags as word`
	`...`
	`can_config_flags = CANSPI_CONFIG_SAMPLE_THRICE and`
	`CANSPI_CONFIG_PHSEG2_PRG_ON and`
	`CANSPI_CONFIG_STD_MSG and`
	`CANSPI_CONFIG_DBL_BUFFER_ON and`
	`CANSPI_CONFIG_VALID_XTD_MSG and`
	`CANSPI_CONFIG_LINE_FILTER_OFF`
	`...`
	`Spi1_Init()`
	`' initialize SPI1 module`
	`CANSPI1Init(1,3,3,3,1,can_config_flags, PORTF, 1, PORTF, 0)`
	`' initialize CANSPI1`

CANSPI1SetBaudRate

Prototype	`sub procedure CANSPI1SetBaudRate(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS as byte)`
Description	Sets the CANSPI1 module baud rate. Due to complexity of the CAN protocol, you can not simply force a bps value. Instead, use this function when the CANSPI1 module is in Config mode. `SAM`, `SEG2PHTS` and `WAKFIL` bits are set according to the `CAN_CONFIG_FLAGS` value. Refer to datasheet for details. Parameters: - `SJW` as defined in CAN controller's datasheet - `BRP` as defined in CAN controller's datasheet - `PHSEG1` as defined in CAN controller's datasheet - `PHSEG2` as defined in CAN controller's datasheet - `PROPSEG` as defined in CAN controller's datasheet - `CAN_CONFIG_FLAGS` is formed from predefined constants (see CANSPI constants)
Requires	The CANSPI1 module must be in Config mode, otherwise the function will be ignored. See CANSPI1SetOperationMode. The CANSPI1 routines are supported only by MCUs with the SPI1 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' set required baud rate and sampling rules`
	`dim can_config_flags as byte`
	`...`
	`CANSPI1SetOperationMode(CANSPI_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CANSPI1 module mast be in`
	`' config mode for baud rate settings)`
	`can_config_flags = CANSPI_CONFIG_SAMPLE_THRICE and`
	`CANSPI_CONFIG_PHSEG2_PRG_ON and`
	`CANSPI_CONFIG_STD_MSG and`
	`CANSPI_CONFIG_DBL_BUFFER_ON and`
	`CANSPI_CONFIG_VALID_XTD_MSG and`
	`CANSPI_CONFIG_LINE_FILTER_OFF`
	`CANSPI1SetBaudRate(1, 1, 3, 3, 1, can_config_flags)`

CANSPI1SetMask

Prototype	`sub procedure CANSPI1SetMask(dim CAN_MASK as byte, dim val as longint, dim CAN_CONFIG_FLAGS as byte)`
Description	Configures mask for advanced filtering of messages. The parameter val is bit-adjusted to the appropriate mask registers. Parameters: - `CAN_MASK`: CANSPI1 module mask number. Valid values: CANSPI_MASK constants (see CANSPI constants) - `val`: mask register value - `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values: `CANSPI_CONFIG_ALL_VALID_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE` and `CANSPI_CONFIG_STD_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE` and `CANSPI_CONFIG_XTD_MSG`. (see CANSPI constants)
Requires	The CANSPI1 module must be in Config mode, otherwise the function will be ignored. See CANSPI1SetOperationMode. The CANSPI1 routines are supported only by MCUs with the SPI1 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' set the appropriate filter mask and message type value`
	`CANSPI1SetOperationMode(CANSPI_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CANSPI1 module must be in config mode`
	`' for mask settings)`
	
	`' Set all B1 mask bits to 1 (all filtered bits are relevant):`
	`' Note that -1 is just a cheaper way to write 0xFFFFFFFF.`
	`' Complement will do the trick and fill it up with ones.`
	`CANSPI1SetMask(CANSPI_MASK_B1, -1, CANSPI_CONFIG_MATCH_MSG_TYPE and CANSPI_CONFIG_XTD_MSG)`

CANSPI1SetFilter

Prototype	`sub procedure CANSPI1SetFilter(dim CAN_FILTER as byte, dim val as longint, dim CAN_CONFIG_FLAGS as byte)`
Description	Configures message filter. The parameter val is bit-adjusted to the appropriate filter registers. Parameters: - `CAN_FILTER`: CANSPI1 module filter number. Valid values: `CANSPI_FILTER` constants (see CANSPI constants) - `val`: filter register value - `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values: `CANSPI_CONFIG_ALL_VALID_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE` and `CANSPI_CONFIG_STD_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE` and `CANSPI_CONFIG_XTD_MSG`. (see CANSPI constants)
Requires	The CANSPI1 module must be in Config mode, otherwise the function will be ignored. See CANSPI1SetOperationMode. The CANSPI1 routines are supported only by MCUs with the SPI1 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' set the appropriate filter value and message type`
	`CANSPI1SetOperationMode(CANSPI_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CANSPI1 module must be in`
	`' config mode for filter settings)`
	
	`' Set id of filter B1_F1 to 3:`
	`CANSPI1SetFilter(CANSPI_FILTER_B1_F1, 3, CANSPI_CONFIG_XTD_MSG)`

CANSPI1Read

Prototype	`sub function CANSPI1Read(dim byref id as longint, dim byref Data as byte[8], dim byref DataLen as byte, dim byref CAN_RX_MSG_FLAGS as byte) as byte`
Returns	- `0` if nothing is received - `0xFFFF` if one of the Receive Buffers is full (message received)
Description	If at least one full Receive Buffer is found, it will be processed in the following way: - Message ID is retrieved and stored to location provided by the id parameter - Message data is retrieved and stored to a buffer provided by the Data parameter - Message length is retrieved and stored to location provided by the DataLen parameter - Message flags are retrieved and stored to location provided by the `CAN_RX_MSG_FLAGS` parameter Parameters: - `id`: message identifier storage address - `Data`: data buffer (an array of bytes up to 8 bytes in length) - `DataLen`: data length storage address. - `CAN_RX_MSG_FLAGS`: message flags storage address
Requires	The CANSPI1 module must be in a mode in which receiving is possible. See CANSPI1SetOperationMode. The CANSPI1 routines are supported only by MCUs with the SPI1 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`dim msg_rcvd, rx_flags, data_len as byte`
	`  data as byte[8]`
	`  msg_id as longint`
	`...`
	`CANSPI1SetOperationMode(CANSPI_MODE_NORMAL,0xFF)' set NORMAL mode`
	`' (CANSPI1 module must be in mode in which receive is possible)`
	`...`
	`rx_flags = 0 ' clear message flags`
	`if (msg_rcvd = CANSPI1Read(msg_id, data, data_len, rx_flags))`
	`then`
	`  ...`
	`end if`

CANSPI1Write

Prototype	`sub function CANSPI1Write(dim id as longint, dim byref Data as byte[ 8] , dim DataLen, CAN_TX_MSG_FLAGS as byte) as byte`
Returns	- `0` if all Transmit Buffers are busy - `0xFFFF` if at least one Transmit Buffer is available
Description	If at least one empty Transmit Buffer is found, the function sends message in the queue for transmission. Parameters: - `id:CAN` message identifier. Valid values: 11 or 29 bit values, depending on message type (standard or extended) - `Data`: data to be sent (an array of bytes up to 8 bytes in length) - `DataLen`: data length. Valid values: 1 to 8 - `CAN_RX_MSG_FLAGS`: message flags
Requires	The CANSPI1 module must be in mode in which transmission is possible. See CANSPI1SetOperationMode. The CANSPI1 routines are supported only by MCUs with the SPI1 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' send message extended CAN message`
	`' with the appropriate ID and data`
	`dim tx_flags as byte`
	`  data as byte[ 8]`
	`  msg_id as longint`
	`...`
	`CANSPI1SetOperationMode(CAN_MODE_NORMAL,0xFF)`
	`' set NORMAL mode (CANSPI1 must be in mode in`
	`' which transmission is possible)`
	
	`tx_flags = CANSPI_TX_PRIORITY_0 and CANSPI_TX_XTD_FRAME`
	`' set message flags`
	`CANSPI1Write(msg_id, data, 2, tx_flags)`

CANSPI2SetOperationMode

Prototype	`sub procedure CANSPI2SetOperationMode(dim mode as byte, dim WAIT as byte)`
Description	Sets the CANSPI module to requested mode. Parameters : - `mode`: CANSPI2 module operation mode. Valid values: `CANSPI_OP_MODE` constants (see CANSPI constants). - `WAIT`: CANSPI2 mode switching verification request. If `WAIT = 0`, the call is non-blocking. The function does not verify if the CANSPI2 module is switched to requested mode or not. The caller must use `CANSPI2GetOperationMode` to verify correct operation mode before performing mode specific operation. If `WAIT <> 0`, the call is blocking – the function won't "return" until the requested mode is set.
Requires	The CANSPI2 routines are supported only by MCUs with the SPI2 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' set the CANSPI2 module into configuration mode`
	`' (wait inside CANSPI2SetOperationMode until this mode is set)`
	`CANSPI2SetOperationMode(CANSPI_MODE_CONFIG, 0xFF)`

CANSPI2GetOperationMode

Prototype	`sub function CANSPI2GetOperationMode() as byte`
Returns	Current operation mode.
Description	The CANSPI2 routines are supported only by the MCUs with SPI2 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' check whether the CANSPI2 module is in Normal mode`
	`' and if it is do something.`
	`if (CANSPI2GetOperationMode() = CANSPI_MODE_NORMAL) then`
	`  ...`
	`end if`

CANSPI2Init

Prototype	`sub procedure CANSPI2Init(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS as byte, dim byref RstPort as word, dim RstPin as word, dim byref CSPort as word, dim CSPin as word)`
Description	Initializes the CANSPI2 module. Stand-Alone CAN controller in the CANSPI module is set to: - Disable CAN capture - Continue CAN operation in Idle mode - Do not abort pending transmissions - Fcan clock : 4*Tcy (Fosc) - Baud rate is set according to the given parameters - CAN mode : Normal - Filter and mask registers IDs are set to zero - Filter and mask message frame type is set according to the `CAN_CONFIG_FLAGS` value `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to the `CAN_CONFIG_FLAGS` value. Parameters: - `SJW` as defined in CAN controller's datasheet - `BRP` as defined in CAN controller's datasheet - `PHSEG1` as defined in CAN controller's datasheet - `PHSEG2` as defined in CAN controller's datasheet - `PROPSEG` as defined in CAN controller's datasheet - `CAN_CONFIG_FLAGS` is formed from predefined constants (see CANSPI constants)
Requires	The CANSPI2 routines are supported only by MCUs with the SPI2 module. The SPI2 module needs to be initialized. See Spi2_Init and the Spi2_Init_Advanced routines. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`dim can_config_flags as word`
	`...`
	`can_config_flags = CANSPI_CONFIG_SAMPLE_THRICE and`
	`CANSPI_CONFIG_PHSEG2_PRG_ON and`
	`CANSPI_CONFIG_STD_MSG and`
	`CANSPI_CONFIG_DBL_BUFFER_ON and`
	`CANSPI_CONFIG_VALID_XTD_MSG and`
	`CANSPI_CONFIG_LINE_FILTER_OFF`
	`...`
	`Spi2_Init() ' initialize SPI`
	`CANSPI2Initialize(1, 1, 3, 3, 1, can_config_flags)`
	`' initialize CANSPI2`

CANSPI2SetBaudRate

Prototype	`sub procedure CANSPI2SetBaudRate(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS as byte)`
Description	Sets the CANSPI2 module baud rate. Due to complexity of the CAN protocol, you can not simply force a bps value. Instead, use this function when the CANSPI2 module is in Config mode. `SAM`, `SEG2PHTS` and `WAKFIL` bits are set according to the `CAN_CONFIG_FLAGS` value. Refer to datasheet for details. Parameters: - `SJW` as defined in CAN controller's datasheet - `BRP` as defined in CAN controller's datasheet - `PHSEG1` as defined in CAN controller's datasheet - `PHSEG2` as defined in CAN controller's datasheet - `PROPSEG` as defined in CAN controller's datasheet - `CAN_CONFIG_FLAGS` is formed from predefined constants (see CANSPI constants)
Requires	The CANSPI2 module must be in Config mode, otherwise the function will be ignored. See CANSPI2SetOperationMode. The CANSPI2 routines are supported only by MCUs with the SPI2 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' set required baud rate and sampling rules`
	`dim can_config_flags as byte`
	`...`
	`CANSPI2SetOperationMode(CANSPI_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CANSPI2 module mast be in config mode`
	`' for baud rate settings)`
	`can_config_flags = CANSPI_CONFIG_SAMPLE_THRICE and`
	`CANSPI_CONFIG_PHSEG2_PRG_ON and`
	`CANSPI_CONFIG_STD_MSG and`
	`CANSPI_CONFIG_DBL_BUFFER_ON and`
	`CANSPI_CONFIG_VALID_XTD_MSG and`
	`CANSPI_CONFIG_LINE_FILTER_OFF`
	`CANSPI2SetBaudRate(1, 1, 3, 3, 1, can_config_flags)`

CANSPI2SetMask

Prototype	`sub procedure CANSPI2SetMask(dim CAN_MASK as byte, dim val as longint, dim CAN_CONFIG_FLAGS as byte)`
Description	Configures mask for advanced filtering of messages. The parameter val is bit-adjusted to the appropriate mask registers. Parameters: - `CAN_MASK`: CANSPI2 module mask number. Valid values: CANSPI_MASK constants (see CANSPI constants) - `val`: mask register value - `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values: `CANSPI_CONFIG_ALL_VALID_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE` and `CANSPI_CONFIG_STD_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE` and `CANSPI_CONFIG_XTD_MSG`. (see CANSPI constants)
Requires	The CANSPI2 module must be in Config mode, otherwise the function will be ignored. See CANSPI2SetOperationMode. The CANSPI2 routines are supported only by MCUs with the SPI2 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' set the appropriate filter mask and message type value`
	`CANSPI2SetOperationMode(CANSPI_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CANSPI2 module must be in config mode`
	`' for mask settings)`
	
	`' Set all B1 mask bits to 1 (all filtered bits are relevant):`
	`' Note that -1 is just a cheaper way to write 0xFFFFFFFF.`
	`' Complement will do the trick and fill it up with ones.`
	`CANSPI2SetMask(CANSPI_MASK_B1, -1, CANSPI_CONFIG_MATCH_MSG_TYPE and CANSPI_CONFIG_XTD_MSG)`

CANSPI2SetFilter

Prototype	`sub procedure CANSPI2SetFilter(dim CAN_FILTER as byte, dim val as longint, dim CAN_CONFIG_FLAGS as byte)`
Description	Configures message filter. The parameter val is bit-adjusted to the appropriate filter registers. Parameters: - `CAN_FILTER`: CANSPI2 module filter number. Valid values: CANSPI_FILTER constants (see CANSPI constants) - `val`: filter register value - `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values: `CANSPI_CONFIG_ALL_VALID_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE` and `CANSPI_CONFIG_STD_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE` and `CANSPI_CONFIG_XTD_MSG`. (see CANSPI constants)
Requires	The CANSPI2 module must be in Config mode, otherwise the function will be ignored. See CANSPI2SetOperationMode. The CANSPI2 routines are supported only by MCUs with the SPI2 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' set the appropriate filter value and message type`
	`CANSPI2SetOperationMode(CANSPI_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (CANSPI2 module must be in config mode`
	`' for filter settings)`
	
	`' Set id of filter B1_F1 to 3:`
	`CANSPI2SetFilter(CANSPI_FILTER_B1_F1, 3, CANSPI_CONFIG_XTD_MSG)`

CANSPI2Read

Prototype	`sub function CANSPI2Read(dim byref id as longint, dim byref Data as byte[8], dim byref DataLen as byte, dim byref CAN_RX_MSG_FLAGS as byte) as byte`
Returns	- `0` if nothing is received - `0xFFFF` if one of the Receive Buffers is full (message received)
Description	If at least one full Receive Buffer is found, it will be processed in the following way: - Message ID is retrieved and stored to location provided by the id parameter - Message data is retrieved and stored to a buffer provided by the Data parameter - Message length is retrieved and stored to location provided by the DataLen parameter - Message flags are retrieved and stored to location provided by the CAN_RX_MSG_FLAGS parameter Parameters: - `id`: message identifier storage address - `data`: data buffer (an array of bytes up to 8 bytes in length) - `dataLen`: data length storage address - `CAN_RX_MSG_FLAGS`: message flags storage address
Requires	The CANSPI2 module must be in a mode in which receiving is possible. See CANSPI2SetOperationMode. The CANSPI2 routines are supported only by MCUs with the SPI2 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' check the CANSPI2 module for received messages.`
	`' If any was received do something.`
	`dim msg_rcvd, rx_flags, data_len as byte`
	`  data as byte[8]`
	`  msg_id as longint`
	`...`
	`CANSPI2SetOperationMode(CANSPI_MODE_NORMAL,0xFF)' set NORMAL mode`
	`' (CANSPI2 module must be in mode in which receive is possible)`
	`...`
	`rx_flags = 0 '`
	`clear message flags`
	`if (msg_rcvd = CANSPI2Read(msg_id, data, data_len, rx_flags))`
	`then`
	`  ...`
	`end if`

CANSPI2Write

Prototype	`sub function CANSPI2Write(dim id as longint, dim byref Data as byte[ 8] , dim DataLen, CAN_TX_MSG_FLAGS as byte) as byte`
Returns	- `0` if all Transmit Buffers are busy - `0xFFFF` if at least one Transmit Buffer is available
Description	If at least one empty Transmit Buffer is found, the function sends message in the queue for transmission. Parameters: - `id`: CAN message identifier. Valid values: 11 or 29 bit values, depending on message type (standard or extended) - `Data`: data to be sent (an array of bytes up to 8 bytes in length) - `DataLen`: data length. Valid values: 1 to 8 - `CAN_RX_MSG_FLAGS`: message flags
Requires	The CANSPI2 module must be in mode in which transmission is possible. See CANSPI2SetOperationMode. The CANSPI2 routines are supported only by MCUs with the SPI2 module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See the example of HW connection at the bottom of this page.

Example	`' send message extended CAN message with the`
	`' appropriate ID and data`
	`dim tx_flags as byte`
	`  data as byte[ 8]`
	`  msg_id as longint`
	`...`
	`CANSPI2SetOperationMode(CAN_MODE_NORMAL,0xFF)`
	`' set NORMAL mode (CANSPI2 must be in mode in`
	`' which transmission is possible)`
	
	`tx_flags = CANSPI_TX_PRIORITY_0 and CANSPI_TX_XTD_FRAME`
	`' set message flags`
	`CANSPI2Write(msg_id, data, 2, tx_flags)`

CANSPI Constants

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

CANSPI_OP_MODE

The CANSPI_OP_MODE constants define CANSPI operation mode. The functions CANSPI1SetOperationMode and CANSPI2SetOperationMode expect one of them as their argument:

```
const
          CANSPI_MODE_BITS   as byte = $E0
          CANSPI_MODE_NORMAL as byte = 0
          CANSPI_MODE_SLEEP  as byte = $20
          CANSPI_MODE_LOOP   as byte = $40
          CANSPI_MODE_LISTEN as byte = $60
          CANSPI_MODE_CONFIG as byte = $80
```

CANSPI_CONFIG_FLAGS

The CANSPI_CONFIG_FLAGS constants define flags related to the CANSPI module configuration. The functions CANSPI1Init, CANSPI2Init, CANSPI1SetBaudRate, CANSPI2SetBaudRate, CANSPI1SetMask, CANSPI2SetMask, CANSPI1SetFilter and CANSPI2SetFilter expect one of them (or a bitwise combination) as their argument:

```
const
    CANSPI_CONFIG_DEFAULT         as byte  = $FF    ' 11111111

    CANSPI_CONFIG_PHSEG2_PRG_BIT as byte  = $01
    CANSPI_CONFIG_PHSEG2_PRG_ON  as byte  = $FF    ' XXXXXXX1
    CANSPI_CONFIG_PHSEG2_PRG_OFF as byte  = $FE    ' XXXXXXX0

    CANSPI_CONFIG_LINE_FILTER_BIT as byte = $02
    CANSPI_CONFIG_LINE_FILTER_ON  as byte = $FF    ' XXXXXX1X
    CANSPI_CONFIG_LINE_FILTER_OFF as byte = $FD    ' XXXXXX0X

    CANSPI_CONFIG_SAMPLE_BIT       as byte = $04
    CANSPI_CONFIG_SAMPLE_ONCE      as byte = $FF    ' XXXXX1XX
    CANSPI_CONFIG_SAMPLE_THRICE    as byte = $FB    ' XXXXX0XX

' continues..
```

```
' ..continued

    CANSPI_CONFIG_MSG_TYPE_BIT     as byte = $08
    CANSPI_CONFIG_STD_MSG          as byte = $FF    ' XXXX1XXX
    CANSPI_CONFIG_XTD_MSG          as byte = $F7    ' XXXX0XXX

    CANSPI_CONFIG_DBL_BUFFER_BIT   as byte = $10
    CANSPI_CONFIG_DBL_BUFFER_ON    as byte = $FF    ' XXX1XXXX
    CANSPI_CONFIG_DBL_BUFFER_OFF   as byte = $EF    ' XXX0XXXX

    CANSPI_CONFIG_MSG_BITS         as byte = $60
    CANSPI_CONFIG_ALL_MSG          as byte = $FF    ' X11XXXXX
    CANSPI_CONFIG_VALID_XTD_MSG    as byte = $DF    ' X10XXXXX
    CANSPI_CONFIG_VALID_STD_MSG    as byte = $BF    ' X01XXXXX
    CANSPI_CONFIG_ALL_VALID_MSG    as byte = $9F    ' X00XXXXX
```

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

```
init  = CANSPI_CONFIG_SAMPLE_THRICE     and
        CANSPI_CONFIG_PHSEG2_PRG_ON     and
        CANSPI_CONFIG_STD_MSG           and
        CANSPI_CONFIG_DBL_BUFFER_ON     and
        CANSPI_CONFIG_VALID_XTD_MSG     and
        CANSPI_CONFIG_LINE_FILTER_OFF
...
CANSPIInit(1, 1, 3, 3, 1, init)    ' initialize CANSPI
```

CANSPI_TX_MSG_FLAGS

CANSPI_TX_MSG_FLAGS are flags related to transmission of a CAN message:

```
const
    CANSPI_TX_PRIORITY_BITS as byte = $03
    CANSPI_TX_PRIORITY_0    as byte = $FC    ' XXXXXX00
    CANSPI_TX_PRIORITY_1    as byte = $FD    ' XXXXXX01
    CANSPI_TX_PRIORITY_2    as byte = $FE    ' XXXXXX10
    CANSPI_TX_PRIORITY_3    as byte = $FF    ' XXXXXX11

    CANSPI_TX_FRAME_BIT     as byte  = $08
    CANSPI_TX_STD_FRAME     as byte  = $FF   ' XXXXX1XX
    CANSPI_TX_XTD_FRAME     as byte  = $F7   ' XXXXX0XX

    CANSPI_TX_RTR_BIT       as byte = $40
    CANSPI_TX_NO_RTR_FRAME  as byte = $FF    ' X1XXXXXX
    CANSPI_TX_RTR_FRAME     as byte = $BF    ' X0XXXXXX
```

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

```
' form value to be used with CANSendMessage:
send_config  = CANSPI_TX_PRIORITY_0     and
               CANSPI_TX_XTD_FRAME      and
               CANSPI_TX_NO_RTR_FRAME
...
CANSPI1Write(id, data, 1, send_config)
```

CANSPI_TX_MSG_FLAGS

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

```
const
    CANSPI_RX_FILTER_BITS  as byte = $07
    CANSPI_RX_FILTER_1     as byte = $00
    CANSPI_RX_FILTER_2     as byte = $01
    CANSPI_RX_FILTER_3     as byte = $02
    CANSPI_RX_FILTER_4     as byte = $03
    CANSPI_RX_FILTER_5     as byte = $04
    CANSPI_RX_FILTER_6     as byte = $05
    CANSPI_RX_OVERFLOW     as byte = $08
    CANSPI_RX_INVALID_MSG  as byte = $10
    CANSPI_RX_XTD_FRAME    as byte = $20
    CANSPI_RX_RTR_FRAME    as byte = $40
    CANSPI_RX_DBL_BUFFERED as byte = $80
```

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

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

CANSPI_MASK

The CANSPI_MASK constants define mask codes. The functions CANSPI1SetMask and CANSPI1SetMask expect one of them as their argument:

```
const
     CANSPI_MASK_B1 as byte = 0
     CANSPI_MASK_B2 as byte = 1
```

CANSPI_FILTER

The CANSPI_FILTER constants define filter codes. The functions CANSPI1SetFilter and CANSPI2SetFilter expects one of them as their argument:

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

Library Example

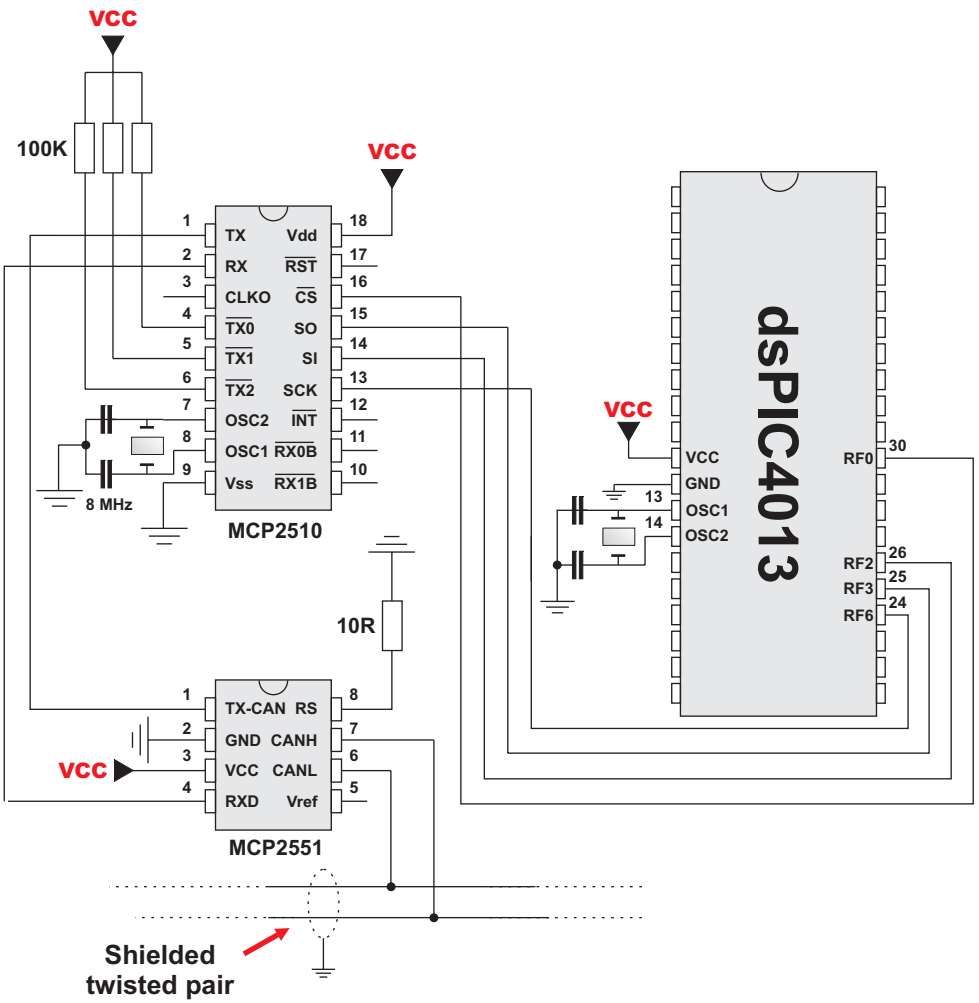
The code is a simple demonstration of CANSPI protocol. This node initiates the communication with the 2nd node by sending some data to its address. The 2nd node responds by sending back the data incremented by 1. This (1st) node then does the same and sends incremented data back to the 2nd node, etc.

```
program CanSPI1st

dim
    Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags, Rx_Data_Len as byte
    RxTx_Data as string[ 8]
    Tx_ID, Rx_ID as longint
    Msg_Rcvd as byte

main:
  ADPCFG = 0xFFFF
  PORTB  = 0
  TRISB  = 0

  Can_Init_Flags  = 0
  Can_Send_Flags  = 0
  Can_Rcv_Flags   = 0

  Can_Send_Flags = CANSPI_TX_PRIORITY_0 and     ' form value to be used
                   CANSPI_TX_XTD_FRAME and     ' with CANSPISendMessage
                   CANSPI_TX_NO_RTR_FRAME

  Can_Init_Flags = CANSPI_CONFIG_SAMPLE_THRICE and  ' form value to be used
                   CANSPI_CONFIG_PHSEG2_PRG_ON and ' with CANSPIInitialize
                   CANSPI_CONFIG_XTD_MSG and
                   CANSPI_CONFIG_DBL_BUFFER_ON and
                   CANSPI_CONFIG_VALID_XTD_MSG

  ' for faster spi communication use Spi_Init_Advanced routine
  Spi1_Init()
  CANSPI1Init(1, 3, 3, 3, 1, Can_Init_Flags, PORTF, 1, PORTF, 0)
  ' initialize external CAN module

  CANSPI1SetOperationMode(CANSPI_MODE_CONFIG,0xFF)  ' set CONFIGURATION mode

' continues ..
```

```
' continued ..

  RxTx_Data[ 0]  = 9

  Tx_ID = 12111

  CANSPI1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags)
  while TRUE
    Msg_Rcvd = CANSPI1Read(Rx_ID, RxTx_Data , Rx_Data_Len,
Can_Rcv_Flags)  ' receive data, if any
    if(((Rx_ID = 3) and (Msg_Rcvd <> 0)) <> 0) then
      PORTB = RxTx_Data[ 0]
      RxTx_Data[ 0]  = RxTx_Data[ 0]  + 1   ' output data at portB
      Delay_ms(5)
      CANSPI1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags)
    end if
  wend
end.

program CanSPI2nd

dim Can_Init_Flags,Can_Send_Flags,Can_Rcv_Flags,Rx_Data_Len as byte
     RxTx_Data as byte[ 8]
     Tx_ID, Rx_ID as longint
     Msg_Rcvd as byte

main:

  ADPCFG = 0xFFFF
  PORTB  = 0
  TRISB  = 0

  Can_Init_Flags = 0
  Can_Send_Flags = 0
  Can_Rcv_Flags  = 0

  Can_Send_Flags = CANSPI_TX_PRIORITY_0 and ' Form value to be used
                   CANSPI_TX_XTD_FRAME  and ' with CANSendMessage
                   CANSPI_TX_NO_RTR_FRAME


' continues...
```

```
' continued ..

  Can_Init_Flags = CANSPI_CONFIG_SAMPLE_THRICE and      ' Form value to be used
                   CANSPI_CONFIG_PHSEG2_PRG_ON and      '  with CANInitialize
                   CANSPI_CONFIG_XTD_MSG        and
                   CANSPI_CONFIG_DBL_BUFFER_ON and
                   CANSPI_CONFIG_VALID_XTD_MSG and
                   CANSPI_CONFIG_LINE_FILTER_OFF


  ' for faster spi communication use Spi_Init_Advanced routine
  Spi1_Init()
  CANSPI1Init(1,3,3,3,1, Can_Init_Flags, PORTF, 1, PORTF, 0) ' initialize CAN

  CANSPI1SetOperationMode(CANSPI_MODE_CONFIG,0xFF)    ' set CONFIGURATION mode

  CANSPI1SetMask(CANSPI_MASK_B1, -1, CANSPI_CONFIG_XTD_MSG)
  ' set all mask1 bits to ones
  CANSPI1SetMask(CANSPI_MASK_B2, -1, CANSPI_CONFIG_XTD_MSG)
  ' set all mask2 bits to ones
  CANSPI1SetFilter(CANSPI_FILTER_B2_F3,12111,CANSPI_CONFIG_XTD_MSG)
  ' set id of filter B2_F3 to 3

  CANSPI1SetOperationMode(CANSPI_MODE_NORMAL,0xFF) ' set NORMAL mode

  Tx_ID = 3

  while TRUE
    Msg_Rcvd = CANSPI1Read(Rx_ID , RxTx_Data , Rx_Data_Len, Can_Rcv_Flags)
    if ((Rx_ID = 12111) and (Msg_Rcvd <> 0)) <> 0 then
      PORTB = RxTx_Data[ 0]                ' output data at portB
      RxTx_Data[ 0]  = RxTx_Data[ 0]  + 1
      CANSPI1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags)
    end if
  wend
end.
```

Hardware Connection



Compact Flash Library

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

In CF card, data is divided into sectors. One sector usually comprises 512 bytes. Routines for file handling, the Cf_Fat routines, are not performed directly but successively through 512B buffer.
Note: Routines for file handling can be used only with FAT16 file system.
Note: Library functions create and read files from the root directory only.
Note: Library functions populate both FAT1 and FAT2 tables when writing to files, but the file data is being read from the FAT1 table only; i.e. there is no recovery if the FAT1 table gets corrupted.

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

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

Library Routines

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

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

Cf_Init

Prototype	`sub procedure Cf_Init(dim byref port_A012 as word, dim pin_A0, pin_A1, pin_A2 as word, dim byref port_RDY as word, dim pin_RDY as word, dim byref port_WE as word, dim pin_WE as word, dim byref port_OE as word, dim pin_OE as word, dim byref port_CD1 as word, dim pin_CD1 as word, dim byref port_CE1 as word, dim pin_CE1 as word, dim byref port_Data as word)`
Description	- Initializes ports appropriately for communication with CF card. Parameters : - `port_A012`: address pins port. - `pin_A2`, `pin_A1`, `pin_A0`: address pins - `port_RDY`: **ready** signal port address - `pin_RDY`: **ready** signal pin - `port_WE`: **write enable** port address - `pin_WE`: **write enable** pin - `port_OE`: **output enable** port address - `pin_OE`: **output enable** pin - `port_CD1`: **chip detect** port address - `pin_CD1`: **chip detect** pin - `port_CE1`: **chip enable** port address - `pin_CE1`: **chip enable** pin - `port_Data`: data port. Pins `<7:0>` are used.

Example	`'-------------- Init for EasydsPIC2`
	`sub procedure Cf_Init_EASYdsPIC2()`
	`  Cf_Init(PORTB,8,9,10, PORTC,15, PORTC,14, PORTC,13, PORTB,12, PORTB,11, PORTB)`
	`end sub`
	
	`'-------------- Init for dsPICPRO`
	`sub procedure Cf_Init_dsPICPRO2()`
	`  Cf_Init(PORTD,8,9,10, PORTG,14, PORTG,12, PORTD,11, PORTG,15, PORTG,13, PORTD)`
	`end sub`

Cf_Detect

Prototype	`sub function CF_Detect() as word`
Returns	- `1` - if CF card was detected - `0` - otherwise
Description	Checks for presence of CF card by reading the chip detect pin. **Important**: The dsPIC30 family MCU and CF card voltage levels are different. The user must ensure that MCU's pin connected to CD line can read CF card Logical One correctly.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`.

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

Cf_Enable

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

Cf_Disable

Prototype	`sub procedure Cf_Disable()`
Returns	Nothing.
Description	Routine disables the device and frees the data lines for other devices. To enable the device again, call Cf_Enable. These two routines in conjunction allow you to free/occupy data line when working with multiple devices.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`.
Example	`' disable compact flash` `Cf_Disable()`

Cf_Read_Init

Prototype	`sub procedure Cf_Read_Init(dim address as longint, dim sectcnt as word)`
Description	Initializes CF card for reading. Parameters : - `address`: the first sector to be prepared for reading operation. - `sectcnt`: number of sectors to be prepared for reading operation.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`.
Example	`' initialize compact flash for reading from sector 590` `Cf_Read_Init(590, 1)`

Cf_Read_Byte

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

Example	`' Read a byte from compact flash:`
	`dim data as byte`
	`...`
	`data = Cf_Read_Byte()`

Cf_Write_Init

Prototype	`sub procedure Cf_Write_Init(dim address as longint, dim sectcnt as word)`
Description	Initializes CF card for writing. Parameters : - `address`: the first sector to be prepared for writing operation. - `sectcnt`: number of sectors to be prepared for writing operation.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`.
Example	`' initialize compact flash for writing to sector 590` `Cf_Write_Init(590, 1)`

Cf_Write_Byte

Prototype	`sub procedure Cf_Write_Byte(dim data as byte)`
Description	Writes a byte to Compact Flash sector buffer location currently pointed to by writing pointers. These pointers will be autoicremented upon reading. When sector buffer is full, its content will be transfered to appropriate flash memory sector. Parameters : - `data`: byte to be written.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`. CF card must be initialized for writing operation. See `Cf_Write_Init`.

Example	`dim data as byte`
	`...`
	`data = 0xAA`
	`Cf_Write_Byte(data)`

Cf_Read_Sector

Prototype	`sub procedure Cf_Read_Sector(dim sector_number as longint, dim byref buffer as byte[ 512] )`
Description	Reads one sector (512 bytes). Read data is stored into buffer provided by the buffer parameter. Parameters : - `sector_number`: sector to be read. - `buffer`: data buffer of at least 512 bytes in length.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`.

Example	`' read sector 22`
	`dim data as byte[ 512]`
	`...`
	`Cf_Read_Sector(22, data)`

Cf_Write_Sector

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

Example	`' write to sector 22`
	`dim data as byte[ 512]`
	`...`
	`Cf_Write_Sector(22, data)`

Cf_Fat_Init

Prototype	`sub function Cf_Fat_Init(dim byref port_A012 as word, dim pin_A0, pin_A1, pin_A2 as word, dim byref port_RDY as word, dim pin_RDY as word, dim byref port_WE as word, dim pin_WE as word, dim byref port_OE as word, dim pin_OE as word, dim byref port_CD1 as word, dim pin_CD1 as word, dim byref port_CE1 as word, dim pin_CE1 as word, dim byref port_Data as word) as word`
Returns	- `0` - if CF card was detected and successfully initialized - `1` - if FAT16 boot sector was not found - `255` - if card was not detected
Description	Initializes CF card, reads CF FAT16 boot sector and extracts data needed by the library. Parameters : - `port_A012`: address pins port. - `pin_A2`, `pin_A1`, `pin_A0`: address pins - `port_RDY`: **ready** signal port address - `pin_RDY`: **ready** signal pin - `port_WE`: **write enable** port address - `pin_WE`: **write enable** pin - `port_OE`: **output enable** port address - `pin_OE`: **output enable** pin - `port_CD1`: **chip detect** port address - `pin_CD1`: **chip detect** pin - `port_CE1`: **chip enable** port address - `pin_CE1`: **chip enable** pin - `port_Data`: data port address. Pins `<7:0>` are used.
Requires	Nothing.

Example	`'--- init the FAT library - dsPICPRO2`
	`if (Cf_Fat_Init(PORTD,8,9,10, PORTG,14, PORTG,12, PORTD,11, PORTG,15, PORTG,13, PORTD) = 0) then`
	`  ...`
	`end if`

Cf_Fat_QuickFormat

Prototype	`sub function Cf_Fat_QuickFormat(dim byref port_A012 as word, dim pin_A0, pin_A1, pin_A2 as word, dim byref port_RDY as word, dim pin_RDY as word, dim byref port_WE as word, dim pin_WE as word, dim byref port_OE as word, dim pin_OE as word, dim byref port_CD1 as word, dim pin_CD1 as word, dim byref port_CE1 as word, dim pin_CE1 as word, dim byref port_Data as word, dim byref cf_fat_label as string[11]) as word`
Returns	- `0` - if CF card was detected, successfully formated and initialized - `1` - if FAT16 format was unseccessful - `255` - if card was not detected
Description	Formats to FAT16 and initializes CF card. Parameters : - `port_A012`: address pins port. - `pin_A2`, `pin_A1`, `pin_A0`: address pins - `port_RDY`: **ready** signal port address - `pin_RDY`: **ready** signal pin - `port_WE`: **write enable** port address - `pin_WE`: **write enable** pin - `port_OE`: **output enable** port address - `pin_OE`: **output enable** pin - `port_CD1`: **chip detect** port address - `pin_CD1`: **chip detect** pin - `port_CE1`: **chip enable** port address - `pin_CE1`: **chip enable** pin - `port_Data`: data port address. Pins `<7:0>` are used. - `cf_fat_label`: volume label (11 characters in length). If less than 11 characters are provided, the label will be padded with spaces. If an empty string is passed, the volume will not be labeled. **Note**: This routine can be used instead or in conjunction with the `Cf_Fat_Init` routine. **Note**: If CF card already contains a valid boot sector, it will remain unchanged (except volume label field) and only FAT and ROOT tables will be erased. Also, the new volume label will be set.

Example	`'--- format and initialize the FAT library - dsPICPRO2`
	`if ( Cf_Fat_QuickFormat(PORTD,8,9,10, PORTG,14, PORTG,12, PORTD,11, PORTG,15, PORTG,13, PORTD, 'mikroE') = 0) then`
	`  ...`
	`end if`

Cf_Fat_Assign

Prototype	`sub function Cf_Fat_Assign(dim byref filename as string[2], dim file_cre_attr as byte) as word`
Returns	- `0` if file does not exist and no new file is created. - `1` if file already exists or file does not exist but a new file is created.
Description	Assigns file for file operations (read, write, delete...). All subsequent file operations will be applied to the assigned file. Parameters : - `filename`: name of the file that should be assigned for file operations. The file name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to the proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between the file name and extension (i.e. "MIKROELETXT" -> MIKROELE.TXT). In this case the last 3 characters of the string are considered to be file extension. - `file_cre_attr`: file creation and attributs flags. Each bit corresponds to the appropriate file attribut:

BIT	MASK	Description
0	0x01	Read Only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume Label
4	0x10	Subdirectory
5	0x20	Archive
6	0x40	Device (internal use only, never found on disk)
7	0x80	File creation flag. If the file does not exist and this flag is set, a new file with specified name will be created.

Note: Long File Names (LFN) are not supported.

Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`.
Example	`' create file with archive attribut if it does not already exist` `Cf_Fat_Assign("MIKRO007.TXT",0xA0)`

Cf_Fat_Reset

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

Example	`dim size as longint`
	`...`
	`Cf_Fat_Reset(size)`

Cf_Fat_Read

Prototype	`sub procedure Cf_Fat_Read(dim byref bdata as byte)`
Description	Reads a byte from currently assigned file opened for reading. Upon function execution file pointers will be set to the next character in the file. Parameters : - `bdata`: buffer to store read byte to. Upon this function execution read byte is returned through this parameter.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. File must be previously assigned. See `Cf_Fat_Assign`. File must be open for reading. See `Cf_Fat_Reset`.

Example	`dim character as byte`
	`...`
	`Cf_Fat_Read(character)`

Cf_Fat_Rewrite

Prototype	`sub procedure Cf_Fat_Rewrite()`
Returns	Nothing.
Description	Opens currently assigned file for writing. If the file is not empty its content will be erased.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. The file must be previously assigned. See `Cf_Fat_Assign`.
Example	`' open file for writing` `Cf_Fat_Rewrite()`

Cf_Fat_Append

Prototype	`sub procedure Cf_Fat_Append()`
Returns	Nothing.
Description	Opens currently assigned file for appending. Upon this function execution file pointers will be positioned after the last byte in the file, so any subsequent file writing operation will start from there.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. File must be previously assigned. See `Cf_Fat_Assign`.
Example	`' open file for appending` `Cf_Fat_Append()`

Cf_Fat_Delete

Prototype	`sub procedure Cf_Fat_Delete()`
Description	Deletes currently assigned file from CF card.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.
Example	`' delete current file` `Cf_Fat_Delete()`

Cf_Fat_Write

Prototype	`sub procedure Cf_Fat_Write(dim byref fdata as byte[ 12] , dim data_len as word)`
Returns	Nothing.
Description	Writes requested number of bytes to currently assigned file opened for writing. Parameters : - `fdata`: data to be written. - `data_len`: number of bytes to be written
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign. File must be open for writing. See Cf_Fat_Rewrite or Cf_Fat_Append.

Example	`dim file_contents as array[ 42]`
	`...`
	`Cf_Fat_Write(file_contents, 42) ' write data to the assigned file`

Cf_Fat_Set_File_Date

Prototype	`sub procedure Cf_Fat_Set_File_Date(dim year as word, dim month as byte, dim day as byte, dim hours as byte, dim mins as byte, dim seconds as byte)`
Returns	Nothing.
Description	Sets the date/time stamp. Any subsequent file writing operation will write this stamp to currently assigned file's time/date attributes. Parameters : - `year`: year attribute. Valid values: 1980-2107 - `month`: month attribute. Valid values: 1-12 - `day`: day attribute. Valid values: 1-31 - `hours`: hours attribute. Valid values: 0-23 - `mins`: minutes attribute. Valid values: 0-59 - `seconds`: seconds attribute. Valid values: 0-59
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign. File must be open for writing. See Cf_Fat_Rewrite or Cf_Fat_Append.
Example	`Cf_Fat_Set_File_Date(2005,9,30,17,41,0)`

Cf_Fat_Get_File_Date

Prototype	`sub procedure Cf_Fat_Get_File_Date(dim byref year as word, dim byref month as byte, dim byref day as byte, dim byref hours as byte, dim byref mins as byte)`
Description	Reads time/date attributes of currently assigned file. Parameters : - `year`: buffer to store year attribute to. Upon function execution year attribute is returned through this parameter. - `month`: buffer to store month attribute to. Upon function execution month attribute is returned through this parameter. - `day`: buffer to store day attribute to. Upon function execution day attribute is returned through this parameter. - `hours`: buffer to store hours attribute to. Upon function execution hours attribute is returned through this parameter. - `mins`: buffer to store minutes attribute to. Upon function execution minutes attribute is returned through this parameter.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`dim year as word`
	`  month, day, hours, mins as byte`
	`...`
	`Cf_Fat_Get_File_Date(year, month, day, hours, mins)`

Cf_Fat_Get_File_Size

Prototype	`sub function Cf_Fat_Get_File_Size() as longint`
Returns	Size of the currently assigned file in bytes.
Description	This function reads size of currently assigned file in bytes.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`dim my_file_size as longint`
	`...`
	`my_file_size = Cf_Fat_Get_File_Size()`

Cf_Fat_Get_Swap_File

Prototype	`sub function Cf_Fat_Get_Swap_File(dim sectors_cnt as longint, dim byref filename as string[11], dim file_attr as byte) as longint`
Returns	- Number of the start sector for the newly created swap file, if there was enough free space on CF card to create file of required size. - `0` - otherwise.
Description	This function is used to create a swap file of predefined name and size on the CF media. If a file with specified name already exists on the media, search for consecutive sectors will ignore sectors occupied by this file. Therefore, it is recommended to erase such file if it exists before calling this function. If it is not erased and there is still enough space for a new swap file, this function will delete it after allocating new memory space for a new swap file. The purpose of the swap file is to make reading and writing to CF media as fast as possible, by using the Cf_Read_Sector() and Cf_Write_Sector() functions directly, without potentially damaging the FAT system. The swap file can be considered as a "window" on the media where the user can freely write/read data. Its main purpose in the mikroBasic's library is to be used for fast data acquisition; when the time-critical acquisition has finished, the data can be re-written into a "normal" file, and formatted in the most suitable way. Parameters: - `sectors_cnt`: number of consecutive sectors that user wants the swap file to have. - `filename`: name of the file that should be assigned for file operations. The file name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to the proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between the file name and extension (i.e. "MIKROELETXT" -> MIKROELE.TXT). In this case the last 3 characters of the string are considered to be file extension. - `file_attr`: file creation and attributs flags. Each bit corresponds to the appropriate file attribut: *' continues on the next page ...*

```
' continued from the previous page ...
```

BIT	MASK	Description
0	0x01	Read Only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume Label
4	0x10	Subdirectory
5	0x20	Archive
6	0x40	Device (internal use only, never found on disk)
7	0x80	Not Used

Note: Long File Names (LFN) are not supported.

Requires

CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`.

Example

```
' Try to create a swap file with archive atribute,
' whose size will be at least 1000 sectors.
' If it succeeds, it sends the No. of start sector over USART
dim size as longint
...
size = Cf_Fat_Get_Swap_File(1000, 'mikroE.txt', 0x20)
if (size <> 0) then
  Usart_Write(0xAA)
  Usart_Write(Lo(size))
  Usart_Write(Hi(size))
  Usart_Write(Higher(size))
  Usart_Write(Highest(size))
  Usart_Write(0xAA)
end if
```

Library Example

This program test routines for reading/writing bytes from/to Compact Flash card and routines for reading/writing Compact Flash card sectors.

```
program Mmc_Fat16_Test

const
  SWAP_FILE_MSG as string[ 15]  = "Swap file at: "

dim
 fat_txt as string[ 20]
 file_contents as string[ 50]

 filename as byte[ 14]              ' File names

 loop1, loop12 as word
 caracter as byte
 i, size as longint

 Buffer as byte[ 512]

sub procedure Glob_Init
 fat_txt = "FAT16 not found"
 file_contents = "XX  CF/SD FAT16 library by Anton Rieckert" + Chr(10) + Chr(0)
 filename = "MIKRO00xTXT"  ' File names
end sub

'M-M-M--------- Creates new file and writes some data to it
sub procedure M_Create_New_File
  filename[ 7]  = "A"
  Cf_Fat_Assign(filename, 0xA0)       ' Will not find file and then create file
  Cf_Fat_Rewrite                      ' To clear file and start with new data
  for loop1 = 1 to 99                 ' We want 5 files on the MMC card
    Uart1_Write_Char(".")
    file_contents[ 0]  = (loop1 div 10)  + 48
    file_contents[ 1]  = (loop1 mod 10)  + 48
    'Uart1_Write_Text(file_contents)
    Cf_Fat_Write(file_contents, 42)    ' write data to the assigned file
  next loop1
end sub'~



' .. continues
```

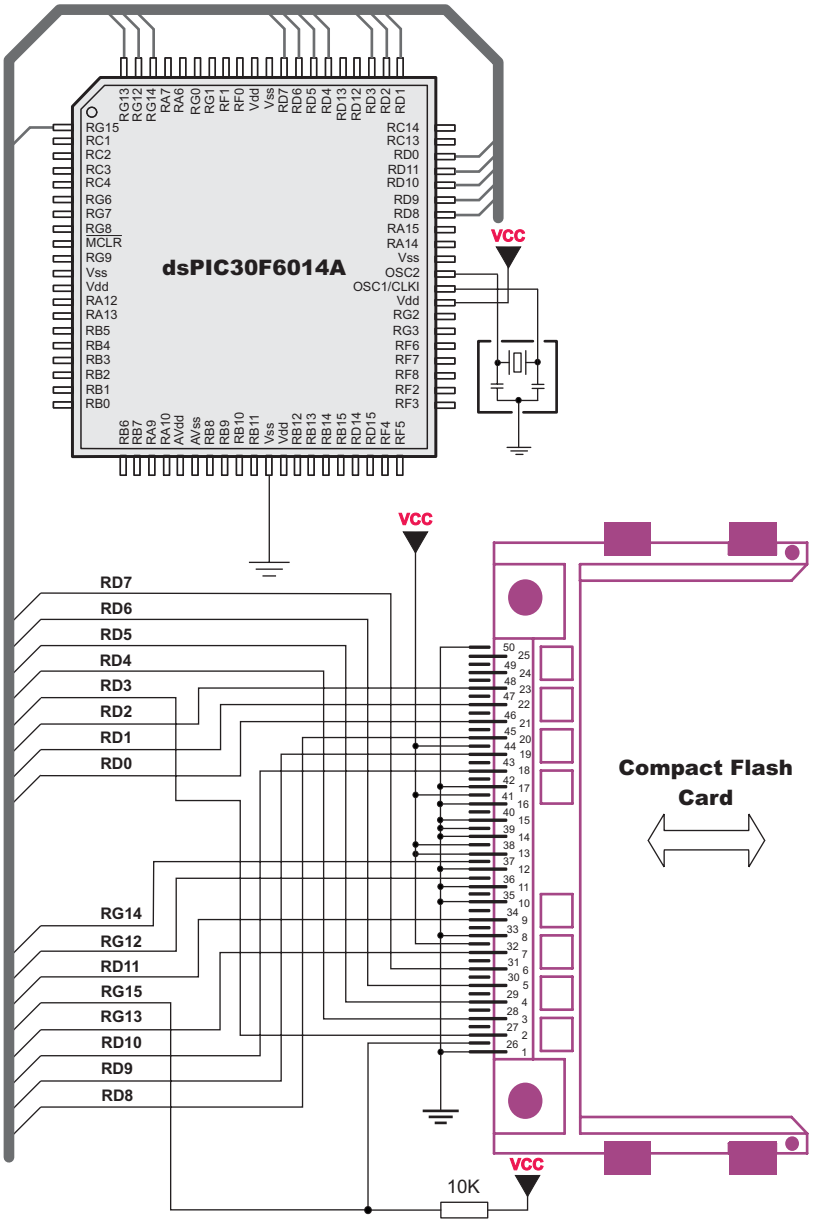
```
' .. continued

'M-M-M--------- Creates many new files and writes data to them
sub procedure M_Create_Multiple_Files
  for loop12 = "B" to "Z"
    Uart1_Write_Char(loop12)          ' signal the progress
    filename[ 7] = loop12             ' set filename
    Cf_Fat_Assign(filename, 0xA0)     ' find existing file or create a new one
    Cf_Fat_Rewrite                    ' To clear file and start with new data
    for loop1 = 1 to 44
      file_contents[ 0] = (loop1 div 10) + 48
      file_contents[ 1] = (loop1 mod 10) + 48
      Cf_Fat_Write(file_contents, 42)   ' write data to the assigned file
    next loop1
  next loop12
end sub'~

'M-M-M--------- Opens an existing file and rewrites it
sub procedure M_Open_File_Rewrite
  filename[ 7] = "C"
  Cf_Fat_Assign(filename, 0)
  Cf_Fat_Rewrite
  for loop1 = 1 to 55
    file_contents[ 0] = loop1 div 10 + 64
    file_contents[ 1] = loop1 mod 10 + 64
    Cf_Fat_Write(file_contents, 42)     ' write data to the assigned file
  next loop1
end sub'~

'M-M-M--------- Opens an existing file and appends data to it
'                (and alters the date/time stamp)
sub procedure M_Open_File_Append
  filename[ 7] = "B"
  Cf_Fat_Assign(filename, 0)
  Cf_Fat_Set_File_Date(2005,6,21,10,35,0)
  Cf_Fat_Append ' Prepare file for append
  Cf_Fat_Write(" for mikroElektronika 2005\n", 27) ' Write data to assigned file
end sub'~

' .. continues
```

```
' .. continued

' M-M-M----Opens an existing file, reads data from it
' and puts it to USART
sub procedure M_Open_File_Read
  filename[ 7] = "B"
  Cf_Fat_Assign(filename, 0)
  Cf_Fat_Reset(size)
  ' To read file, sub procedure returns size of file
  for i = 1 to size
    Cf_Fat_Read(caracter)
    Uart1_Write_Char(caracter)  ' Write data to USART
  next i
end sub   '~

'M-M-M--------- Deletes a file. If file doesn't exist,
'it will first be created
'and then deleted.
sub procedure M_Delete_File
  filename[ 7] = "F"
  Cf_Fat_Assign(filename, 0)
  Cf_Fat_Delete
end sub    '~

' M-M-M--------- Tests whether file exists,
' and if so sends its creation date
' and file size via USART
sub procedure M_Test_File_Exist(dim fLetter as byte)
  dim
    fsize as longint
    year as word
    month1, day, hour1, minute1 as byte
    outstr as byte[ 12]

  filename[ 7] = fLetter
  if (Cf_Fat_Assign(filename, 0) <> 0) then
    '--- file has been found - get its date
    Cf_Fat_Get_File_Date(year, month1, day, hour1, minute1)
    WordToStr(year, outstr)
    Uart1_Write_Text(outstr)
    WordToStr(month1, outstr)
    Uart1_Write_Text(outstr)
    WordToStr(day, outstr)
    Uart1_Write_Text(outstr)
    WordToStr(hour1, outstr)
    Uart1_Write_Text(outstr)
    WordToStr(minute1, outstr)
    Uart1_Write_Text(outstr)

' .. continues
```

```
' .. continued

    '--- get file size
    fsize = Cf_Fat_Get_File_Size
    LongIntToStr(longint(fsize), outstr)
    Uart1_Write_Text(outstr)
  else
    '--- file was not found - signal it
    Uart1_Write_Char(0x55)
    Delay_ms(1000)
    Uart1_Write_Char(0x55)
  end if
end sub  '~

'-------------- Tries to create a swap file, whose
' size will be at least 100
' sectors (see Help for details)
sub procedure M_Create_Swap_File
  dim i as word

  for i=0 to 511
    Buffer[ i]  = i
  next i

  size = Cf_Fat_Get_Swap_File(5000, "mikroE.txt", 0x20)
  ' see help on this function for details

  if (size <> 0) then
    LongIntToStr(size, fat_txt)
    Uart1_Write_Text(fat_txt)

    for i=0 to 4999
      Cf_Write_Sector(size, Buffer)
      Inc(size)
      Uart1_Write_Char(".")
    next i
  end if
end sub  '~

' .. continues
```

```
' .. continued
main:
      Glob_Init
      '--- prepare PORTB for signalling
      ADPCFG = 0xFFFF
      PORTB = 0
      TRISB = 0
      '--- set up USART for the file read
      Uart1_Init(19200)
      Uart1_Write_Text("start")
      '--- init the FAT library
      '** dsPIC30F601x, dsPICPro3
      ' use fat16 quick format instead of init routine for format
      CF_fat_Init(PORTG, 0, 1, 2,
                      PORTG, 9,
                      PORTG, 8,
                      PORTG, 7,
                      PORTG, 6,
                      PORTG, 3,
                      PORTD)' data port
                      = 0 then
      '**
          Delay_ms(500)
          '--- Test start
          PORTB = 0x0005
          '--- Test routines. Uncomment them one-by-one for test
          M_Create_New_File
          M_Create_Multiple_Files
          Uart1_Write_Char(10)

          M_Open_File_Rewrite
          M_Open_File_Append
          M_Delete_File
          M_Create_Swap_File
          Uart1_Write_Char(10)

          M_Open_File_Read
          Uart1_Write_Char(10)
          M_Test_File_Exist("F")   ' this file will not exist here
          Uart1_Write_Char(10)
          M_Test_File_Exist("B")   ' this file will exist here
          Uart1_Write_Char(10)
      else
          Uart1_Write_Text(fat_txt)
      end if
      '--- Test termination
      PORTB = 0xFFFF
end.
```

Hardware Connection



DSP Library

mikroBasic for dsPIC30/33 and PIC24 includes libraries for DSP engine.

Library Routines :

```
FIR_Radix
IIR_Radix
FFT
IFFT
BitReverseComplex
Vector_Set
VectorPower
Vector_Subtract
VectorScale
Vector_Negate
Vector_Multiply
Vector_Min
Vector_Max
Vector_Dot
Vector_Correlate
Vector_Convolve
Vector_Add
Matrix_Transponse
Matrix_Subtract
Matrix_Scale
Matrix_Multiply
Matrix_Add
```

FIR_Radix

Prototype	`sub function FIR_Radix(dim FilterOrder as Word, dim ptrCoeffs as LongInt, dim BuffLength as Word, dim ptrInput as Word, dim Index as Word) as Word`
Description	This function applies FIR filter to ptrInput. Input samples must be in Y data space. - `FilterOrder` is order of the filter + 1. - `ptrCoeffs` is address of filter coeffitients in program memory. - `BuffLength` represents number of samples ptrInput points to. - `ptrInput` is address of input samples. - `Index` index of current sample.

Returns	`sum(k=0..N-1)(coef[k]*input[N-k]) - Current sample of processed signal(B[n])`
	`N - buffer length`
	`k - Current index`

IIR_Radix

Prototype	`sub function IIR_Radix(dim BScale as Integer, dim AScale as Integer, dim ptrB as Word, dim ptrA as Word, dim FilterOrder as Word, dim ptrInput as Word, dim Input_Len as Word, dim ptrOutput as Word, dim Index as Word) as Word`
Description	This function applies IIR filter to ptrInput. Input and output samples must be in Y data space. - `AScale` A Scale factor - `BScale` B Scale factor - `ptrB` Address of B coefficients (In program memory) - `ptrA` Address of A coefficients (In program memory) - `FilterOrder` is order of the filter + 1. - `ptrInput` is address of input samples. - `Input_Len` represents number of samples ptrInput points to. - `ptrOutput` is address of output samples. Output length is equal to Input length. - `Index` index of current sample.
Returns	`y[n]=sum(k=0..N)(Acoef[k]*x[n-k]) - sum(k=1..M)(Bcoef[k]*y[n-k])`

FFT

Prototype	`sub procedure FFT(dim log2N as word, dim TwiddleFactorsAddress as LongInt, dim byref Samples as word[1024])`
Description	Function applies FFT transformation to input samples, input samples must be in Y data space. - `N` - buffer length (must be the power of 2). - `TwiddleFactorsAddress` is address of costant array which contains complex twiddle factors.The array is expected to be in program memory. - `Samples` array of input samples. Upon completion complex array of FFT samples is placed in the Samples parameter.
Operation	**F(k) = 1/N*sum_n (f(n)*WN(kn)), WN(kn) = exp[-(j*2*pi*k*n)/N]** Fn - array of complex input samples n in {0, 1,... , N-1}, and k in {0, 1,... , N-1}, with N = 2^m, m element of Z. WN - TwiddleFactors The amplitude of current FFT sample is calculated as: **F[k]=sqrt(Re[k]^2+ Im[k]^2)**
Note	Complex array of FFT samples is placed in Samples parameter. Input Samples are arranged in manner Re,Im,Re,Im... (where Im is always zero). Output samples are arranged in the same manner but Im parts are different from zero. Output samples are symmetrical (First half of output samples (index from 0 to N/2) is identical to the second half of output samples(index from N/2 to N). Input data is a complex vector such that the magnitude of the real and imaginary parts of each of its elements is less than 0.5. If greater or equal to this value the results could produce saturation. Note that the output values are scaled by a factor of 1/N, with N the length of the FFT. input is expected in natural ordering, while output is produced in bit reverse ordering.

IFFT

Prototype	`sub procedure IFFT(dim log2N as word, dim TwiddleFactorsAddress as LongInt, dim byref FftSamples as word[1024])`
Description	Function applies IFFT transformation to input samples, input samples must be in Y data space. - `N` - buffer length (must be the power of 2). - `TwiddleFactorsAddress` is address of costant array which contains complex twiddle factors.The array is expected to be in program memory. - `Samples` array of input samples. Upon completion complex array of IFFT samples is placed in the Samples parameter.
Operation	**f(k) = 1/N*sum_n (F(n)*WN(kn)), WN(kn) = exp[(j*2*pi*k*n)/N]** Fn - array of complex input samples n in {0, 1,... , N-1}, and k in {0, 1,... , N-1}, with N = 2^m, m element of Z. WN - TwiddleFactors
Note	Complex array of IFFT samples is placed in Samples parameter. Input Samples are arranged in manner Re,Im,Re,Im... (where Im is always zero). Input data is a complex vector such that the magnitude of the real and imaginary parts of each of its elements is less than 0.5. If greater or equal to this value the results could produce saturation. Note that the output values are scaled by a factor of 1/N, with N the length of the IFFT. Input is expected in bit reverse ordering, while output is produced in natural ordering.

BitReverseComplex

Prototype	`sub procedure BitReverseComplex(dim log2N as word, dim byref ReIm as word[1024])`
Description	This function does Complex (in-place) Bit Reverse re-organization. - `N` - buffer length (must be the power of 2). - `ReIm` - Output Sample(from FFT).
Note	Input samples must be in Y data space.

Vector_Set

Prototype	`sub procedure Vector_Set(dim byref input as array[1024] of word, dim size, value as word);`
Description	Sets size elements of input to value, starting from the first element. Size must be > 0. Length of input is limited by available ram.

VectorPower

Prototype	`sub function VectorPower(dim N as word, dim byref Vector as word[1024]) as word`
Description	Function returns result of power value (powVal) in radix point 1.15
Operation	**powVal = sum (srcV[n] * srcV[n])** with n in {0, 1,... , numElems-1}
Input	`N` = number of the elements in vector(s) (numElems) `Vector` = ptr to source vector (srcV)
Note	AccuA used, not restored CORCON saved, used, restored

Vector_Subtract

Prototype	`sub procedure Vector_Subtract(dim byref dest, v1, v2 as word[1024], dim numElems as word)`
Description	This procedure does substraction of two vectors. `numElems` must be less or equal to minimum size of two vectors. - `v1` - First Vector - `v2` - Second Vector - `dest` - Result Vector
Operation	**dstV[n] = srcV1[n] - srcV2[n]** with n in {0, 1,... , numElems-1}
Note	AccuA used, not restored. CORCON saved, used, restored.

VectorScale

Prototype	`sub procedure VectorScale(dim N as word, dim ScaleValue as integer, dim byref SrcVector, DestVector as word[1024])`
Description	This procedure does vector scaling with scale value. - `N` - Buffer length - `SrcVector` - original vector - `DestVector` - scaled vector - `ScaleValue` - Scale Value
Operation	**dstV[n] = sclVal * srcV[n],** with n in {0, 1,... , numElems-1}
Note	AccuA used, not restored. CORCON saved, used, restored

Vector_Negate

Prototype	`sub procedure Vector_Negate(dim byref srcVector, DestVector as word[1024], dim numElems as word)`
Description	This procedure does negation of vector. - `srcVector` - Original vector - `destVector` - Result vector - `numElems` - Number of Elements
Operation	**dstV[n] = (-1)*srcV1[n] + 0,** 0 <= n < numElems
Note	Negate of 0x8000 is 0x7FFF. AccuA used, not restored. CORCON saved, used, restored.

Vector_Multiply

Prototype	`sub procedure Vector_Multiply(dim byref v1, v2, dest as word[1024], dim numElems as word)`
Description	This procedure does multiplication of two vectors. numElems must be less or equal to minimum size of two vectors. - `v1` - First Vector - `v2` - Second Vector - `dest` - Result Vector
Operation	**dstV[n] = srcV1[n] * srcV2[n]** with n in {0, 1,... , numElems-1}
Note	AccuA used, not restored. CORCON saved, used, restored

Vector_Min

Prototype	`sub function Vector_Min(dim byref Vector as word[1024], dim numElems as word, dim byref MinIndex as word) as word`
Description	This function find min. value in vector. - `Vector` - Original vector. - `numElems` - Number of elements - `MinIndex` - Index of minimum value
Operation	**minVal = min {srcV[n], n in {0, 1,...numElems-1}** if srcV[i] = srcV[j] = minVal, and i < j, then minIndex = j
Returns	minimum value (minVal)

Vector_Max

Prototype	`sub function Vector_Max(dim byref Vector as word[1024], dim numElems as word, dim byref MaxIndex as word) as word`
Description	This function find max. value in vector. - `Vector` - Original vector. - `numElems` - Number of elements - `MaxIndex` - Index of maximum value
Operation	**maxVal = max {srcV[n], n in {0, 1,...numElems-1} }** if srcV[i] = srcV[j] = maxVal, and i < j, then maxIndex = j
Returns	maximum value (maxVal)

Vector_Dot

Prototype	`sub function Vector_Dot(dim byref v1, v2 as word[1024], dim numElems as word) as word`
Description	Procedure calculates vector dot product. - `v1` - First vector. - `v2` - Second vector - `numElems` - Number of elements
Operation	**dotVal = sum (srcV1[n] * srcV2[n]),** with n in {0, 1,... , numElems-1}
Note	AccuA used, not restored. CORCON saved, used, restored.

Vector_Correlate

Prototype	`sub procedure Vector_Correlate(dim byref v1, v2, dest as word[1024], dim numElemsV1, numElemsV2 as word)`
Description	Procedure calculates Vector correlation (using convolution). - `v1` - First vector. - `v2` - Second vector - `numElemsV1` - Number of first vector elements - `numElemsV2` - Number of second vector elements - `dest` - Result vector
Operation	**r[n] = sum_(k=0:N-1){x[k]*y[k+n]},** where: x[n] defined for 0 <= n < N, y[n] defined for 0 <= n < M, (M <= N), r[n] defined for 0 <= n < N+M-1.

Vector_Convolve

Prototype	`sub procedure Vector_Convolve(dim byref v1, v2, dest as word[1024], dim numElemsV1, numElemsV2 as word)`
Description	Procedure calculates Vector using convolution. - `v1` - First vector. - `v2` - Second vector - `numElemsV1` - Number of first vector elements - `numElemsV2` - Number of second vector elements - `dest` - Result vector
Operation	y[n] = sum_(k=0:n){x[k]*h[n-k]}, 0 <= n < M y[n] = sum_(k=n-M+1:n){x[k]*h[n-k]}, M <= n < N y[n] = sum_(k=n-M+1:N-1){x[k]*h[n-k]}, N <= n < N+M-1
Note	AccuA used, not restored. CORCON saved, used, restored.

Vector_Add

Prototype	`sub procedure Vector_Add(dim byref dest, v1, v2 as word[256], dim numElems as word)`
Description	Procedure calculates vector addition. - `v1` - First vector. - `v2` - Second vector - `numElemsV1` - Number of vector elements - `dest` - Result vector
Operation	**dstV[n] = srcV1[n] + srcV2[n]**, with n in {0, 1,... , numElems-1}
Note	AccuA used, not restored. CORCON saved, used, restored.

Matrix_Transponse

Prototype	`sub procedure Matrix_Transponse(dim byref src, dest as word[1024], dim numRows, numCols as word)`
Description	Procedure does matrix transposition. `src` - Original matrix. `dest` - Result matrix `numRows` - Number of matrix rows `numCols` - Number of matrix columns
Operation	**dstM[i][j] = srcM[j][i]**

Matrix_Subtract

Prototype	`sub procedure Matrix_Subtract(dim byref src1, src2, dest as word[1024], dim numRows, numCols as word)`
Description	Procedure does matrix substraction. - `src1` - First matrix. - `src2` - Second matrix - `dest` - Result matrix - `numRows` - Number of matrix rows - `numCols` - Number of matrix columns
Operation	**dstM[i][j] = srcM1[i][j] - srcM2[i][j]**
Note	AccuA used, not restored. AccuB used, not restored. CORCON saved, used, restored.

Matrix_Scale

Prototype	`sub procedure Matrix_Scale(dim ScaleValue as word, dim byref src1, dest as word[ 1024] , dim numRows, numCols as word)`
Description	Procedure does matrix scale. - `ScaleValue` - Scale Value - `src1` - Original matrix - `dest` - Result matrix - `numRows` - Number of matrix rows - `numCols` - Number of matrix columns
Operation	**dstM[i][j] = sclVal * srcM[i][j]**
Note	AccuA used, not restored. CORCON saved, used, restored.

Matrix_Multiply

Prototype	`sub procedure Matrix_Multiply(dim byref src1, src2, dest as word[ 256] , dim numRows1, numCols2, numCols1Rows2 as word)`
Description	Procedure does matrix multiply. - `src1` - First Matrix - `src2` - Second Matrix - `dest` - Result Matrix - `numRows1` - Number of first matrix rows - `numCols2` - Number of second matrix columns - `numCols1Rows2` - Number of first matrix columns and second matrix rows
Operation	**dstM[i][j] = sum_k(srcM1[i][k]*srcM2[k][j])**, with i in {0, 1, ..., numRows1-1} j in {0, 1, ..., numCols2-1} k in {0, 1, ..., numCols1Rows2-1}
Note	AccuA used, not restored. CORCON saved, used, restored.

Matrix_Add

Prototype	`sub procedure Matrix_Add(dim byref src1, src2, dest as word[ 1024] , dim numRows, numCols as word)`
Description	Procedure does matrix addition. - `src1` - First Matrix - `src2` - Second Matrix - `dest` - Result Matrix - `numRows1` - Number of first matrix rows - `numCols2` - Number of second matrix columns
Operation	**dstM[i][j] = srcM1[i][j] + srcM2[i][j]**
Note	AccuA used, not restored. CORCON saved, used, restored.

ECAN Library

mikroBasic for dsPIC30/33 and PIC24 provides a library (driver) for working with the dsPIC33FJ and pic24HJ ECAN module.

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

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

ECAN supports two message formats:

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

ECAN message format and DMA RAM buffer definiton can be found in the `ECan_Defs.dbas` header file located in the ECAN project folder. Read this file carefully and make appropriate adjustments for mcu in use. Also, if a new project is to be created, this file has to be copied, adjusted and included into the project with corresponding Search Path updating.

Note: ECAN buffers are located in DMA RAM, so two DMA channels are used for message transfer, one for each direction (ECAN->DMA RAM, DMA RAM->ECAN). See the ECAN1DmaChannelInit and ECAN2DmaChannelInit routines.

Note: Consult CAN standard about CAN bus termination resistance.

```
ECAN1DmaChannelInit
ECAN1SetOperationMode
ECAN1GetOperationMode
ECAN1Initialize
ECAN1SelectTxBuffers
ECAN1FilterDisable
ECAN1FilterEnable
ECAN1SetBufferSize
ECAN1SetBaudRate
ECAN1SetMask
ECAN1SetFilter
ECAN1Read
ECAN1Write

ECAN2DmaChannelInit
ECAN2SetOperationMode
ECAN2GetOperationMode
ECAN2Initialize
ECAN2SelectTxBuffers
ECAN2FilterDisable
ECAN2FilterEnable
ECAN2SetBufferSize
ECAN2SetBaudRate
ECAN2SetMask
ECAN2SetFilter
ECAN2Read
```

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

- RegsToECAN1ID
- ECAN1IDToRegs
- RegsToECAN2ID
- ECAN2IDToRegs

ECAN1DmaChannelInit

Prototype	`sub function ECAN1DmaChannelInit(dim DmaChannel as word, dim ChannelDir as word, dim DmaRamBuffAdd as word) as word`
Returns	- `0` - if dma channel parameter is valid - `0x0001` - if dma channel is already in use (busy) - `0xFFFF` - if dma channel parameter is invalid
Description	The function preforms initialization of the DMA module for ECAN. Parameters: - `DmaChannel`: DMA Channel number. Valid values: 0..7. - `ChannelDir`: transfer direction. Valid values: 1 (dma ram to peripheral) and 0 (peripheral to dma ram). - `DmaRamBuffAdd`: DMA RAM buffer address. DMA RAM location is MCU dependent, refer to datasheet for valid address range.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.
Example	`' channel 0 will transfer 8 words from dma ram at 0x4000 to ECAN1` `ECAN1DmaChannelInit(0, 1, 0x4000)`

ECAN1SetOperationMode

Prototype	`sub procedure ECAN1SetOperationMode(dim mode as word, dim WAIT as word)`
Description	Sets the ECAN1 module to requested mode. Parameters : - `mode`: ECAN1 module operation mode. Valid values: `ECAN_OP_MODE` constants. See ECAN constants. - `WAIT`: ECAN1 mode switching verification request. If `WAIT = 0`, the call is non-blocking. The function does not verify if the ECAN1 module is switched to requested mode or not. The caller must use ECAN1GetOperationMode to verify correct operation mode before performing mode specific operation. If `WAIT <> 0`, the call is blocking – the function won't "return" until the requested mode is set and no additional verification is necessary.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.

Example	`' set the ECAN1 module into configuration mode (wait inside`
	`' ECAN1SetOperationMode until this mode is set)`
	`ECAN1SetOperationMode(ECAN_MODE_CONFIG, 0xFF)`

ECAN1GetOperationMode

Prototype	`sub function ECAN1GetOperationMode() as word`
Returns	Current operation mode.
Description	The function returns current operation mode of the ECAN1 module. Check the `ECAN_OP_MODE` constants (see ECAN constants) or device datasheet for operation mode codes.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.

Example	`' check whether the ECAN1 module is in Normal mode`
	`' and if it is do something.`
	`if (ECAN1GetOperationMode() = ECAN_MODE_NORMAL) then`
	`  ...`
	`end if`

ECAN1Initialize

Prototype	`sub procedure ECAN1Initialize(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, ECAN_CONFIG_FLAGS as word)`
Description	Initializes the ECAN1 module. The internal ECAN1 module is set to: - Disable ECAN capture - Continue ECAN operation in Idle mode - Abort all pending transmissions - Clear all transmit control registers - Fcan clock : Fcy (Fosc/2) - Baud rate is set according to given parameters - ECAN mode is set to Normal - Filter and mask registers remain unchanged The `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to the `ECAN_CONFIG_FLAGS` value. Parameters: - `SJW` as defined in MCU's datasheet (ECAN1 Module) - `BRP` as defined in MCU's datasheet (ECAN1 Module) - `PHSEG1` as defined in MCU's datasheet (ECAN1 Module) - `PHSEG2` as defined in MCU's datasheet (ECAN1 Module) - `PROPSEG` as defined in MCU's datasheet (ECAN1 Module) - `ECAN_CONFIG_FLAGS` ECAN module configuration flags. Each bit corresponds to the appropriate ECAN module parameter. Should be formed out of predefined ECAN flag constants. See ECAN constants. **Note**: ECAN mode NORMAL will be set on exit.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.

Example	`ecan_config_flags = ECAN_CONFIG_SAMPLE_THRICE and`
	`ECAN_CONFIG_PHSEG2_PRG_ON and`
	`ECAN_CONFIG_XTD_MSG and`
	`ECAN_CONFIG_MATCH_MSG_TYPE and`
	`ECAN_CONFIG_LINE_FILTER_OFF`
	`ECAN1Initialize(1, 3, 3, 3, 1, ecan_config_flags)`
	`' initialize the ECAN1 module`

ECAN1SelectTxBuffers

Prototype	`sub function ECAN1SelectTxBuffers(dim txselect as word) as word`
Returns	- `0` - if input parameter is valid - `0xFFFF` - if input parameter is invalid
Description	The function designates the ECAN1 module's transmit buffers. Parameters: - `txselect`: transmit buffer select. By setting bits in the txselect lower byte corresponding buffers are enabled for transmition. The ECAN module supports up to 8 transmit buffers. Also, by clearing bits in the txselect lower byte corresponding buffers are enabled for reception.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN1 module must be initialized. See the ECAN1Initialize routine.
Example	`' Buffers 0 and 2 are enabled for transmition:` `ECAN1SelectTxBuffers(0x0005)`

ECAN1FilterDisable

Prototype	`sub procedure ECAN1FilterDisable(dim fltdis as word)`
Description	The function disables receive filters. Parameters: - `fltdis`: filter disable selection parameter. Each bit corresponds to the appropriate filter. By setting bit, the corresponding filter is to be disabled.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN1 module must be initialized. See the ECAN1Initialize routine.
Example	`' Filters 0, 4, 8, 12 are to be disabled:` `ECAN1FilterDisable(0x1111)`

ECAN1FilterEnable

Prototype	`sub procedure ECAN1FilterEnable(dim flten as word)`
Description	The function enables receive filters. Parameters: - `flten`: filter enable selection parameter. Each bit corresponds to the appropriate filter. By setting bit, the corresponding filter will be enabled.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN1 module must be initialized. See the ECAN1Initialize routine.
Example	`' Filters 0, 4, 8, 12 are to be enabled:` `ECAN1FilterEnable(0x1111)`

ECAN1SetBufferSize

Prototype	`sub function ECAN1SetBufferSize(dim Ecan1BuffSize as word) as word`
Returns	- `0` - if input parameter is valid - `0xFFFF` - if input parameter is invalid
Description	The function configures the total number of receive and transmit buffers in DMA RAM. Parameters: - `Ecan1BuffSize`: Number of ECAN1 DMA RAM receive and transmit buffers. Valid values: 4, 6, 8, 12, 16, 24, 32. Each buffer is 16 bytes long. **Note**: The same value should be used for DMA RAM buffer definition in the `ECan_Defs.dbas` header file located in the ECAN project folder.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN1 module must be initialized. See the ECAN1Initialize routine.
Example	`' DMA RAM will have 16 rx+tx buffers` `ECAN1SetBufferSize(16)`

ECAN1SetBaudRate

Prototype	`sub procedure ECAN1SetBaudRate(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, ECAN_CONFIG_FLAGS as word)`
Description	Sets ECAN1 module baud rate. Due to complexity of the ECAN protocol, you can not simply force the bps value. Instead, use this function when ECAN1 is in Config mode. Refer to datasheet for details. The `SAM`, `SEG2PHTS` and `WAKFIL` bits are set according to the `ECAN_CONFIG_FLAGS` value. Parameters: - `SJW` as defined in MCU's datasheet (ECAN1 Module) - `BRP` as defined in MCU's datasheet (ECAN1 Module) - `PHSEG1` as defined in MCU's datasheet (ECAN1 Module) - `PHSEG2` as defined in MCU's datasheet (ECAN1 Module) - `PROPSEG` as defined in MCU's datasheet (ECAN1 Module) - `ECAN_CONFIG_FLAGS` ECAN module configuration flags. Each bit corresponds to the appropriate ECAN module parameter. Should be formed out of predefined ECAN flag constants. See ECAN constants
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.The ECAN1 module must be in Config mode, otherwise the function will be ignored. See ECAN1SetOperationMode.

Example	`' set required baud rate and sampling rules`
	`dim ecan_config_flags as word`
	`...`
	`ECAN1SetOperationMode(ECAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (ECAN1 module must be in`
	`' config mode for baud rate settings)`
	
	`ecan_config_flags = ECAN_CONFIG_SAMPLE_THRICE and`
	`' Form value to be used`
	`ECAN_CONFIG_PHSEG2_PRG_ON and`
	`' with ECAN1SetBaudRate`
	`ECAN_CONFIG_XTD_MSG and`
	`ECAN_CONFIG_MATCH_MSG_TYPE and`
	`ECAN_CONFIG_LINE_FILTER_OFF`
	
	`ECAN1SetBaudRate(1, 3, 3, 3, 1, ecan_config_flags)`
	`' set ECAN1 module baud rate`

ECAN1SetMask

Prototype	`sub procedure ECAN1SetMask(dim ECAN_MASK as word, dim val as longint, dim ECAN_CONFIG_FLAGS as word)`
Description	The function configures appropriate mask for advanced message filtering. Parameters: - `ECAN_MASK`: ECAN module mask number. Valid values: `ECAN_MASK` constants. See ECAN constants - `val`: mask register value. This value is bit-adjusted to appropriate buffer mask registers - `ECAN_CONFIG_FLAGS`: selects type of messages to filter. Valid values: `ECAN_CONFIG_ALL_VALID_MSG`, `ECAN_CONFIG_MATCH_MSG_TYPE` and `ECAN_CONFIG_STD_MSG`, `ECAN_CONFIG_MATCH_MSG_TYPE` and `ECAN_CONFIG_XTD_MSG`. (see ECAN constants).
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN1 module must be in Config mode, otherwise the function will be ignored. See ECAN1SetOperationMode.

Example	`' set appropriate filter mask and message type value`
	`' ECAN1SetOperationMode(ECAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (ECAN1 module must be in`
	`' config mode for mask settings)`
	
	`' Set all mask0 bits to 1 (all filtered bits are relevant):`
	`' Note that -1 is just a cheaper way to write 0xFFFFFFFF.`
	`' Complement will do the trick and fill it up with ones.`
	`ECAN1SetMask(ECAN_MASK_0, -1, ECAN_CONFIG_MATCH_MSG_TYPE and ECAN_CONFIG_XTD_MSG)`

ECAN1SetFilter

Prototype	`sub procedure ECAN1SetFilter(dim ECAN_FILTER as word, dim val as longint, dim ECAN_FILTER_MASK as word, dim ECAN_FILTER_RXBUFF as word, dim ECAN_CONFIG_FLAGS as word)`
Description	The function configures and enables appropriate message filter. Parameters: - `ECAN_FILTER`: ECAN module filter number. Valid values: `ECAN_FILTER` constants. See ECAN constants - `val`: filter register value. This value is bit-adjusted to appropriate filter registers - `ECAN_FILTER_MASK`: mask register corresponding to filter. Valid values: `ECAN_MASK` constants. See ECAN constants - `ECAN_FILTER_RXBUFF`: receive buffer corresponding to filter. Valid values: `ECAN_RX_BUFFER` constants. See ECAN constants - `ECAN_CONFIG_FLAGS`: selects type of messages to filter. Valid values: `ECAN_CONFIG_XTD_MSG` and `ECAN_CONFIG_STD_MSG`. See ECAN constants.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN1 module must be in Config mode, otherwise the function will be ignored. See ECAN1SetOperationMode.

Example	`' set appropriate filter value and message type`
	`ECAN1SetOperationMode(ECAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (ECAN1 module must be in config mode`
	`' for filter settings)`
	
	`' Set id of filter 10 to 3, mask2, receive buffer 7,`
	`' extended messages:`
	`ECAN1SetFilter(ECAN_FILTER_10, 3, ECAN_MASK_2, ECAN_RX_BUFFER_7,`
	`ECAN_CONFIG_XTD_MSG)`

ECAN1Read

Prototype	`sub function ECAN1Read(dim byref id as longint, dim byref data as byte[ 8] , dim byref dataLen as word, dim byref ECAN_RX_MSG_FLAGS as word) as word`
Returns	- `0` if none of Receive Buffers is full - `0xFFFF` if at least one of Receive Buffers is full (message received)
Description	If at least one full Receive Buffer is found, it will be processed in the following way: - Message ID is retrieved and stored to location pointed by the id pointer - Message data is retrieved and stored to array pointed by the data pointer - Message length is retrieved and stored to location pointed by the dataLen pointer - Message flags are retrieved and stored to location pointed by the `ECAN_RX_MSG_FLAGS` pointer Parameters: - `id`: message identifier address - `data`: an array of bytes up to 8 bytes in length - `dataLen`: data length address - `ECAN_RX_MSG_FLAGS`: message flags address. For message receive flags format refer to the `ECAN_RX_MSG_FLAGS` constants (see ECAN constants)
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN1 module must be in a mode in which receiving is possible. See ECAN1SetOperationMode.

Example	`' check the ECAN1 module for received messages.`
	`' If any was received do something.`
	`dim msg_rcvd, rx_flags, data_len as word`
	`  data as byte[ 8]`
	`  msg_id as longint`
	`...`
	`ECAN1SetOperationMode(ECAN_MODE_NORMAL,0xFF)`
	`' set NORMAL mode (ECAN1 module must be in a mode in which`
	`' receiving is possible)`
	`...`
	`rx_flags = 0`
	`' clear message flags`
	`if (msg_rcvd = ECAN1Read(msg_id, data, data_len, rx_flags)) then`
	`  ...`
	`end if`

ECAN1Write

Prototype	`sub function ECAN1Write(dim id as longint, dim byref Data as byte[ 8] , dim DataLen, ECAN_TX_MSG_FLAGS as word) as word`
Returns	- `0` if all Transmit Buffers are busy - `0xFFFF` if at least one Transmit Buffer is empty and available for transmition
Description	If at least one empty Transmit Buffer is found, the function sends message in the queue for transmission. Parameters: - `id`: ECAN1 message identifier. Valid values: all 11 or 29 bit values, depending on message type (standard or extended) - `Data`: data to be sent - `DataLen`: data length. Valid values: 0..8 - `ECAN_TX_MSG_FLAGS`: message flags. Valid values: `ECAN_TX_MSG_FLAGS` constants. See ECAN constants.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN1 module must be in a mode in which transmission is possible. See ECAN1SetOperationMode.

Example	`' send message extended ECAN message with appropriate ID and data`
	`dim tx_flags as word`
	`  data as byte[ 8]`
	`  msg_id as longint`
	`...`
	`ECAN1SetOperationMode(ECAN_MODE_NORMAL,0xFF)  ' set NORMAL mode`
	`' (ECAN1 must be in a mode in which transmission is possible)`
	
	`tx_flags = ECAN_TX_PRIORITY_0  and`
	`  ECAN_TX_XTD_FRAME  and`
	`  ECAN_TX_NO_RTR_FRAME`
	`' set message flags`
	`ECAN1Write(msg_id, data, 1, tx_flags)`

ECAN2DmaChannelInit

Prototype	`sub function ECAN2DmaChannelInit(dim DmaChannel as word, dim ChannelDir as word, dim DmaRamBuffAdd as word) as word`
Returns	- `0` - if dma channel parameter is valid - `0x0001` - if dma channel already in use (busy) - `0xFFFF` - if dma channel parameter is invalid
Description	The function preforms initialization of the DMA module for ECAN. Parameters: - `DmaChannel`: DMA Channel number. Valid values: 0..7. - `ChannelDir`: transfer direction. Valid values: 1 (dma ram to peripheral) and 0 (peripheral to dma ram). - `DmaRamBuffAdd`: DMA RAM buffer address. DMA RAM location is MCU dependent, refer to datasheet for valid address range.
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.
Example	`'channel 0 will transfer 8 words from dma ram at 0x4000 to ECAN2` `ECAN2DmaChannelInit(0, 1, 0x4000)`

ECAN2SetOperationMode

Prototype	`sub procedure ECAN2SetOperationMode(dim mode as word, dim WAIT as word)`
Description	Sets the ECAN2 module to requested mode. Parameters : - `mode`: ECAN2 module operation mode. Valid values: `ECAN_OP_MODE` constants. See ECAN constants. - `WAIT`: ECAN2 mode switching verification request. If `WAIT = 0`, the call is non-blocking. The function does not verify if the ECAN2 module is switched to requested mode or not. The caller must use `ECAN2GetOperationMode` to verify correct operation mode before performing mode specific operation. If `WAIT <> 0`, the call is blocking – the function won't "return" until the requested mode is set and no additional verification is necessary.
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.

Example	`' set the ECAN2 module into configuration mode (wait inside`
	`' ECAN2SetOperationMode until this mode is set)`
	`ECAN2SetOperationMode(ECAN_MODE_CONFIG, 0xFF)`

ECAN2Initialize

Prototype	`sub procedure ECAN2Initialize(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, ECAN_CONFIG_FLAGS as word)`
Description	Initializes the ECAN2 module. The internal ECAN2 module is set to: - Disable ECAN capture - Continue ECAN operation in Idle mode - Abort all pending transmissions - Clear all transmit control registers - Fcan clock : Fcy (Fosc/2) - Baud rate is set according to given parameters - ECAN mode is set to Normal - Filter and mask registers remain unchanged The `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to the `ECAN_CONFIG_FLAGS` value. Parameters: - `SJW` as defined in MCU's datasheet (ECAN2 Module) - `BRP` as defined in MCU's datasheet (ECAN2 Module) - `PHSEG1` as defined in MCU's datasheet (ECAN2 Module) - `PHSEG2` as defined in MCU's datasheet (ECAN2 Module) - `PROPSEG` as defined in MCU's datasheet (ECAN2 Module) - `ECAN_CONFIG_FLAGS` ECAN module configuration flags. Each bit corresponds to the appropriate ECAN module parameter. Should be formed out of predefined ECAN flag constants. See ECAN constants
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.

Example	`dim ecan_config_flags as word`
	`...`
	`ecan_config_flags = ECAN_CONFIG_SAMPLE_THRICE and`
	`ECAN_CONFIG_PHSEG2_PRG_ON and`
	`ECAN_CONFIG_XTD_MSG and`
	`ECAN_CONFIG_MATCH_MSG_TYPE and`
	`ECAN_CONFIG_LINE_FILTER_OFF`
	
	`ECAN2Initialize(1, 3, 3, 3, 1, ecan_config_flags)`
	`' initialize the ECAN2 module`

ECAN2SelectTxBuffers

Prototype	`sub function ECAN2SelectTxBuffers(dim txselect as word) as word`
Returns	- `0` - if input parameter is valid - `0xFFFF` - if input parameter is invalid
Description	The function designates ECAN2 module's transmit buffers. Parameters: - `txselect`: transmit buffer select. By setting bits in the txselect lower byte corresponding buffers are enabled for transmition. ECAN supports up to 8 transmit buffers. Also, by clearing bits in the txselect lower byte corresponding buffers are enabled for reception.
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN2 module must be initialized. See the ECAN2Initialize routine.
Example	`' Buffers 0 and 2 are enabled for transmition:` `ECAN2SelectTxBuffers(0x0005)`

ECAN2FilterDisable

Prototype	`sub procedure ECAN2FilterDisable(dim fltdis as word)`
Description	The function disables receive filters. Parameters: - `fltdis`: filter disable selection parameter. Each bit corresponds to appropriate filter. By setting bit, the corresponding filter will be disabled.
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN2 module must be initialized. See the ECAN2Initialize routine.
Example	`' Filters 0, 4, 8, 12 are to be disabled:` `ECAN2FilterDisable(0x1111)`

ECAN2FilterEnable

Prototype	`sub procedure ECAN2FilterEnable(dim flten as word)`
Description	The function enables receive filters. Parameters: - `flten`: filter enable selection parameter. Each bit corresponds to appropriate filter. By setting bit, the corresponding filter will be enabled.
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN2 module must be initialized. See the ECAN2Initialize routine.
Example	`' Filters 0, 4, 8, 12 are to be enabled:` `ECAN2FilterEnable(0x1111)`

ECAN2SetBufferSize

Prototype	`sub function ECAN2SetBufferSize(dim Ecan2BuffSize as word) as word`
Returns	- `0` - if input parameter is valid - `0xFFFF` - if input parameter is invalid
Description	The function configures the overall number of receive and transmit buffers in DMA RAM. Parameters: - `Ecan2BuffSize`: Number of ECAN2 DMA RAM receive and transmit buffers. Valid values: 4, 6, 8, 12, 16, 24, 32. Each buffer is 16 bytes long. **Note**: The same value should be used for DMA RAM buffer definition in the `ECan_Defs.dbas` header file located in the ECAN project folder.
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.The ECAN2 module must be initialized. See the ECAN2Initialize routine.
Example	`' DMA RAM will have 16 rx+tx buffers` `ECAN2SetBufferSize(16)`

ECAN2SetBaudRate

Prototype	`sub procedure ECAN2SetBaudRate(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, ECAN_CONFIG_FLAGS as word)`
Description	Sets ECAN2 module baud rate. Due to complexity of the ECAN protocol, you can not simply force the bps value. Instead, use this function when ECAN2 is in Config mode. Refer to datasheet for details. The `SAM`, `SEG2PHTS` and `WAKFIL` bits are set according to the `ECAN_CONFIG_FLAGS` value. Parameters: - `SJW` as defined in MCU's datasheet (ECAN2 Module) - `BRP` as defined in MCU's datasheet (ECAN2 Module) - `PHSEG1` as defined in MCU's datasheet (ECAN2 Module) - `PHSEG2` as defined in MCU's datasheet (ECAN2 Module) - `PROPSEG` as defined in MCU's datasheet (ECAN2 Module) - The `ECAN_CONFIG_FLAGS` ECAN module configuration flags. Each bit corresponds to the appropriate ECAN module parameter. Should be formed out of predefined ECAN flag constants. See ECAN constants.
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN2 module must be in Config mode, otherwise the function will be ignored. See ECAN2SetOperationMode.

Example	`' set required baud rate and sampling rules`
	`dim ecan_config_flags as word`
	`...`
	`ECAN2SetOperationMode(ECAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode (ECAN2 module mast be in config mode for`
	`' baud rate settings)`
	
	`ecan_config_flags = ECAN_CONFIG_SAMPLE_THRICE and`
	`' Form value to be used`
	`ECAN_CONFIG_PHSEG2_PRG_ON and`
	`' with ECAN2SetBaudRate`
	`ECAN_CONFIG_XTD_MSG and`
	`ECAN_CONFIG_MATCH_MSG_TYPE and`
	`ECAN_CONFIG_LINE_FILTER_OFF`
	
	`ECAN2SetBaudRate(1, 3, 3, 3, 1, ecan_config_flags)`
	`' set the ECAN2 module baud rate`

ECAN2SetMask

Prototype	`sub procedure ECAN2SetMask(dim ECAN_MASK as word, dim val as longint, dim ECAN_CONFIG_FLAGS as word)`
Description	The function configures appropriate mask for advanced message filtering. Parameters: - `ECAN_MASK`: ECAN module mask number. Valid values: `ECAN_MASK` constants. See ECAN constants - `val`:mask register value. This value is bit-adjusted to appropriate buffer mask registers - `ECAN_CONFIG_FLAGS`: selects type of messages to filter. Valid values: `ECAN_CONFIG_ALL_VALID_MSG`, `ECAN_CONFIG_MATCH_MSG_TYPE` and `ECAN_CONFIG_STD_MSG`, `ECAN_CONFIG_MATCH_MSG_TYPE` and `ECAN_CONFIG_XTD_MSG`. (see ECAN constants)
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.The ECAN2 module must be in Config mode, otherwise the function will be ignored. See ECAN2SetOperationMode.

Example	`' set appropriate filter mask and message type value`
	`ECAN2SetOperationMode(ECAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode`
	`' (ECAN2 module must be in config mode for mask settings)`
	
	`' Set all mask0 bits to 1 (all filtered bits are relevant):`
	`' Note that -1 is just a cheaper way to write 0xFFFFFFFF.`
	`' Complement will do the trick and fill it up with ones.`
	`ECAN2SetMask(ECAN_MASK_0, -1, ECAN_CONFIG_MATCH_MSG_TYPE and ECAN_CONFIG_XTD_MSG)`

ECAN2SetFilter

Prototype	`sub procedure ECAN2SetFilter(dim ECAN_FILTER as word, dim val as longint, dim ECAN_FILTER_MASK as word, dim ECAN_FILTER_RXBUFF as word, dim ECAN_CONFIG_FLAGS as word)`
Description	The function configures and enables appropriate message filter. Parameters: - `ECAN_FILTER`: ECAN module filter number. Valid values: `ECAN_FILTER` constants. See ECAN constants - val: filter register value. This value is bit-adjusted to appropriate filter registers - `ECAN_FILTER_MASK`: mask register corresponding to filter. Valid values: `ECAN_MASK` constants. See ECAN constants - `ECAN_FILTER_RXBUFF`: receive buffer corresponding to filter. Valid values: `ECAN_RX_BUFFER` constants. See ECAN constants - `ECAN_CONFIG_FLAGS`: selects type of messages to filter. Valid values: `ECAN_CONFIG_XTD_MSG` or `ECAN_CONFIG_STD_MSG`. See ECAN constants.
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN2 module must be in Config mode, otherwise the function will be ignored. See ECAN2SetOperationMode.

Example	`' set appropriate filter value and message type`
	`ECAN2SetOperationMode(ECAN_MODE_CONFIG,0xFF)`
	`' set CONFIGURATION mode`
	`' (ECAN2 module must be in config mode for filter settings)`
	
	`' Set id of filter 10 to 3, mask2,`
	`' receive buffer 7, extended messages:`
	`ECAN2SetFilter(ECAN_FILTER_10, 3, ECAN_MASK_2, ECAN_RX_BUFFER_7, ECAN_CONFIG_XTD_MSG)`

ECAN2Read

Prototype	`sub function ECAN2Read(dim byref id as longint, dim byref data as byte[ 8] , dim byref dataLen as word, dim byref ECAN_RX_MSG_FLAGS as word) as word`
Returns	- `0` if none of Receive Buffers is full - `0xFFFF` if at least one of Receive Buffers is full (message received)
Description	If at least one full Receive Buffer is found, it will be processed in the following way: - Message ID is retrieved and stored to location pointed by the id pointer - Message data is retrieved and stored to array pointed by the data pointer - Message length is retrieved and stored to location pointed by the dataLen pointer - Message flags are retrieved and stored to location pointed by the `ECAN_RX_MSG_FLAGS` pointer Parameters: - `id`: message identifier address - `data`: an array of bytes up to 8 bytes in length - `dataLen`: data length address - `ECAN_RX_MSG_FLAGS`: message flags address. For message receive flags format refer to the `ECAN_RX_MSG_FLAGS` constants (see ECAN constants)
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN2 module must be in a mode in which receiving is possible. See ECAN2SetOperationMode.

Example	`' check the ECAN2 module for received messages.`
	`' If any was received do something.`
	`dim msg_rcvd, rx_flags, data_len as word`
	`  data as byte[ 8]`
	`  msg_id as longint`
	`...`
	`ECAN2SetOperationMode(ECAN_MODE_NORMAL,0xFF)  ' set NORMAL mode`
	`'(ECAN2 module must be in a mode in which receiving is possible)`
	`...`
	`rx_flags = 0 ' clear message flags`
	`if (msg_rcvd = ECAN2Read(msg_id, data, data_len, rx_flags)) then`
	`  ...`
	`end if`

ECAN2Write

Prototype	`sub function ECAN2Write(dim id as longint, dim byref Data as byte[8], dim DataLen, ECAN_TX_MSG_FLAGS as word) as word`
Returns	- `0` if all Transmit Buffers are busy - `0xFFFF` if at least one Transmit Buffer is empty and available for transmition
Description	If at least one empty Transmit Buffer is found, the function sends message in the queue for transmission. Parameters: - `id`: ECAN2 message identifier. Valid values: all 11 or 29 bit values, depending on message type(standard or extended) - `Data`: data to be sent - `DataLen`: data length. Valid values: 0..8 - `ECAN_TX_MSG_FLAGS`: message flags. Refer to the `ECAN_TX_MSG_FLAGS` constants. See ECAN constants
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. The microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus. The ECAN2 module must be in a mode in which receiving is possible. See ECAN2SetOperationMode.

Example	`' send message extended ECAN message with appropriate ID and data`
	`dim tx_flags as word`
	`  data as byte[8]`
	`  msg_id as longint`
	`...`
	`ECAN2SetOperationMode(ECAN_MODE_NORMAL,0xFF)  ' set NORMAL mode`
	`'(ECAN2 must be in a mode in which transmission is possible)`
	
	`tx_flags = ECAN_TX_PRIORITY_0 and`
	`  ECAN_TX_XTD_FRAME  and`
	`  ECAN_TX_NO_RTR_FRAME  ' set message flags`
	`ECAN2Write(msg_id, data, 1, tx_flags)`

ECAN Constants

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

ECAN_OP_MODE

The ECAN_OP_MODE constants define ECAN operation mode. The functions ECAN1SetOperationMode and ECAN2SetOperationMode expect one of them as their argument:

```
const
    ECAN_MODE_BITS          = 0x00E0
    ECAN_MODE_NORMAL        = 0x00
    ECAN_MODE_DISABLE       = 0x01
    ECAN_MODE_LOOP          = 0x02
    ECAN_MODE_LISTEN        = 0x03
    ECAN_MODE_CONFIG        = 0x04
    ECAN_MODE_LISTEN_ALL    = 0x07
```

ECAN_CONFIG_FLAGS

The ECAN_CONFIG_FLAGS constants define flags related to the ECAN module configuration. The functions ECAN1Initialize, ECAN2Initialize, ECAN1SetBaudRate and ECAN2SetBaudRate expect one of them (or a bitwise combination) as their argument:

```
const
    ECAN_CONFIG_DEFAULT           = 0xFF    ' 11111111

    ECAN_CONFIG_PHSEG2_PRG_BIT    = 0x01
    ECAN_CONFIG_PHSEG2_PRG_ON     = 0xFF    ' XXXXXXX1
    ECAN_CONFIG_PHSEG2_PRG_OFF    = 0xFE    ' XXXXXXX0

    ECAN_CONFIG_LINE_FILTER_BIT   = 0x02
    ECAN_CONFIG_LINE_FILTER_ON    = 0xFF    ' XXXXXX1X
    ECAN_CONFIG_LINE_FILTER_OFF   = 0xFD    ' XXXXXX0X

    ECAN_CONFIG_SAMPLE_BIT        = 0x04
    ECAN_CONFIG_SAMPLE_ONCE       = 0xFF    ' XXXXX1XX
    ECAN_CONFIG_SAMPLE_THRICE     = 0xFB    ' XXXXX0XX

' continues..
```

```
' continued..

    ECAN_CONFIG_MSG_TYPE_BIT      = 0x08
    ECAN_CONFIG_STD_MSG            = 0xFF    ' XXXX1XXX
    ECAN_CONFIG_XTD_MSG            = 0xF7    ' XXXX0XXX

    ECAN_CONFIG_MATCH_TYPE_BIT    = 0x20
    ECAN_CONFIG_ALL_VALID_MSG     = 0xDF    ' XX0XXXXX
    ECAN_CONFIG_MATCH_MSG_TYPE    = 0xFF    ' XX1XXXXX
```

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

```
init   = ECAN_CONFIG_SAMPLE_THRICE     and
         ECAN_CONFIG_PHSEG2_PRG_ON     and
         ECAN_CONFIG_STD_MSG            and
         ECAN_CONFIG_MATCH_MSG_TYPE    and
         ECAN_CONFIG_LINE_FILTER_OFF
...
ECAN1Initialize(1, 1, 3, 3, 1, init)    ' initialize ECAN1
```

ECAN_TX_MSG_FLAGS

ECAN_TX_MSG_FLAGS are flags related to transmission of ECAN message. The functions ECAN1Write and ECAN2Write expect one of them (or a bitwise combination) as their argument:

```
const
    ECAN_TX_PRIORITY_BITS   = 0x03
    ECAN_TX_PRIORITY_0      = 0xFC    ' XXXXXX00
    ECAN_TX_PRIORITY_1      = 0xFD    ' XXXXXX01
    ECAN_TX_PRIORITY_2      = 0xFE    ' XXXXXX10
    ECAN_TX_PRIORITY_3      = 0xFF    ' XXXXXX11
    ECAN_TX_FRAME_BIT       = 0x08
    ECAN_TX_STD_FRAME       = 0xFF    ' XXXXX1XX
    ECAN_TX_XTD_FRAME       = 0xF7    ' XXXXX0XX
    ECAN_TX_RTR_BIT         = 0x40
    ECAN_TX_NO_RTR_FRAME    = 0xFF    ' X1XXXXXX
    ECAN_TX_RTR_FRAME       = 0xBF    ' X0XXXXXX
```

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

```
' form value to be used with CANSendMessage:
send_config   = ECAN_TX_PRIORITY_0 and
                ECAN_TX_XTD_FRAME  and
                ECAN_TX_NO_RTR_FRAME
...
ECAN1SendMessage(id, data, 1, send_config)
```

ECAN_RX_MSG_FLAGS

ECAN_RX_MSG_FLAGS are flags related to reception of ECAN message. If a particular bit is set then corresponding meaning is TRUE otherwise it will be FALSE.

```
const
    ECAN_RX_FILTER_BITS  = 0x000F ' Use this to access filter
                                  ' bits
    ECAN_RX_FILTER_0     = 0x00   ' filter0 match
    ECAN_RX_FILTER_1     = 0x01   ' filter1 match
    ECAN_RX_FILTER_2     = 0x02   ' ...
    ECAN_RX_FILTER_3     = 0x03
    ECAN_RX_FILTER_4     = 0x04
    ECAN_RX_FILTER_5     = 0x05
    ECAN_RX_FILTER_6     = 0x06
    ECAN_RX_FILTER_7     = 0x07
    ECAN_RX_FILTER_8     = 0x08
    ECAN_RX_FILTER_9     = 0x09
    ECAN_RX_FILTER_10    = 0x0A
    ECAN_RX_FILTER_11    = 0x0B
    ECAN_RX_FILTER_12    = 0x0C
    ECAN_RX_FILTER_13    = 0x0D
    ECAN_RX_FILTER_14    = 0x0E   ' ...
    ECAN_RX_FILTER_15    = 0x0F   ' filter15 match

    ECAN_RX_OVERFLOW     = 0x10   ' Set if Overflowed else cleared
    ECAN_RX_INVALID_MSG  = 0x20   ' Set if invalid else cleared
    ECAN_RX_XTD_FRAME    = 0x40
    ' Set if XTD message else cleared
    ECAN_RX_RTR_FRAME    = 0x80
    ' Set if RTR message else cleared
```

You may use bitwise **and** to extract received message status. For example:

```
if (MsgFlag and ECAN_RX_OVERFLOW <> 0) then
  ...
  ' Receiver overflow has occurred.
  ' We have lost our previous message.
end if
```

ECAN_MASK

The ECAN_MASK constants define mask codes. The functions ECAN1SetMask and ECAN2SetMask expect one of them as their argument:

```
const
    ECAN_MASK_0   = 0
    ECAN_MASK_1   = 1
    ECAN_MASK_2   = 2
```

ECAN_FILTER

The ECAN_FILTER constants define filter codes. The functions ECAN1SetFilter and ECAN2SetFilter expects one of them as their argument:

```
const
    ECAN_FILTER_0    =  0
    ECAN_FILTER_1    =  1
    ECAN_FILTER_2    =  2
    ECAN_FILTER_3    =  3
    ECAN_FILTER_4    =  4
    ECAN_FILTER_5    =  5
    ECAN_FILTER_6    =  6
    ECAN_FILTER_7    =  7
    ECAN_FILTER_8    =  8
    ECAN_FILTER_9    =  9
    ECAN_FILTER_10   = 10
    ECAN_FILTER_11   = 11
    ECAN_FILTER_12   = 12
    ECAN_FILTER_13   = 13
    ECAN_FILTER_14   = 14
    ECAN_FILTER_15   = 15
```

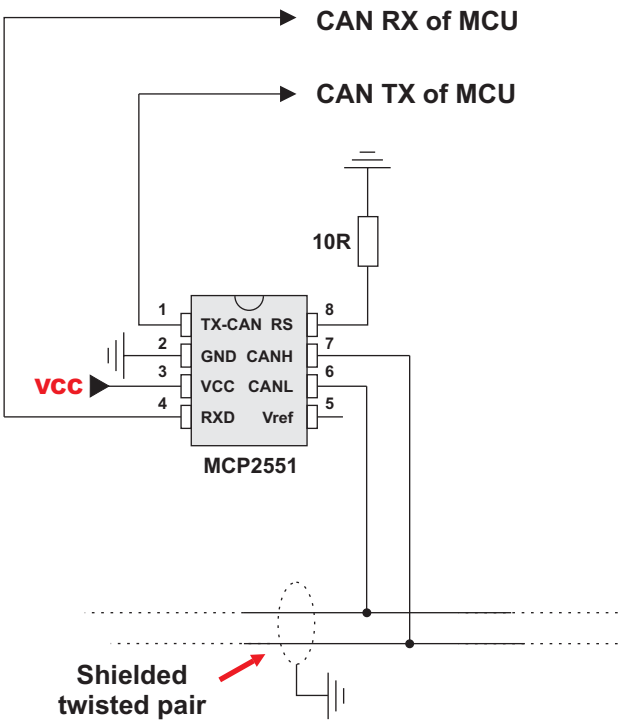
Library Example

The code is a simple demonstration of the ECAN protocol.

```
program ECAN_1st

include "ECan_Defs"

dim Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags as word
                                                        ' can flags
    Rx_Data_Len  as word     ' received data length in bytes
    RxTx_Data    as byte[ 8] ' can rx/tx data buffer
    Msg_Rcvd     as word     ' reception flag
    Tx_ID, Rx_ID as longint  ' can rx and tx ID

sub procedure C1Interrupt() org 0x005A    ' ECAN event iterrupt
      IFS2.3 = 0                   ' clear ECAN interrupt flag
      if(C1INTF.0 <> 0) then    ' was it tx interrupt?
    C1INTF.0 = 0         ' if yes clear tx interrupt flag
  end if

  if(C1INTF.1 <> 0) then   ' was it rx interrupt?
            C1INTF.1 = 0 ' if yes clear rx interrupt flag
      end if
end sub

main:
  ' Set PLL : Fosc = ((Fin/PLLPRE)*PLLDIV)/PLLPOST
  ' (((10MHz/2)*32)/4) = 20MHz
  ' refer the pic24 family datasheet for more details
  CLKDIV = CLKDIV and 0xFFE0   'CLKDIVbits.PLLPRE = 0
  PLLFBD = 0x1E                 'PLLFBDbits.PLLDIV = 0x1E
  CLKDIV = CLKDIV and 0xFF3F   'CLKDIVbits.PLLPOST = 1
  CLKDIV = CLKDIV or 0x00C0

  AD1PCFGH = 0xFFFF                '
  AD1PCFGL = 0xFFFF                ' all ports digital I/O
  AD2PCFGL = 0xFFFF                '

  '{* Clear Interrupt Flags *}

      IFS0 = 0
      IFS1 = 0
      IFS2 = 0
      IFS3 = 0
      IFS4 = 0

' continues..
```

```
' continued..

  '{* Enable ECAN1 Interrupt *}

      IEC2.3    = 1   ' enable ECAN1 interrupts
      C1INTE.0 = 1   ' enable ECAN1 tx interrupt
      C1INTE.1 = 1   ' enable ECAN1 rx interrupt

  PORTB     = 0       ' clear PORTB
  TRISB     = 0       ' set PORTB as output,
                       ' for received message data displaying

  Can_Init_Flags = 0                                    '
  Can_Send_Flags = 0                                    ' clear flags
  Can_Rcv_Flags  = 0                                    '

  Can_Send_Flags = ECAN_TX_PRIORITY_0 and ' Form value to be used
                     ECAN_TX_XTD_FRAME and '  with CANSendMessage
                     ECAN_TX_NO_RTR_FRAME

  Can_Init_Flags = ECAN_CONFIG_SAMPLE_THRICE and
                    ' Form value to be used
                     ECAN_CONFIG_PHSEG2_PRG_ON and
                    '  with CANInitialize
                     ECAN_CONFIG_XTD_MSG and
                     ECAN_CONFIG_MATCH_MSG_TYPE and
                     ECAN_CONFIG_LINE_FILTER_OFF

  RxTx_Data[ 0]  = 9 ' set initial data to be sent
  ECAN1DmaChannelInit(0, 1, @ECAN1RxTxRAMBuffer)
  ' init dma channel 0 for
  ' dma to ECAN peripheral transfer
  ECAN1DmaChannelInit(2, 0, @ECAN1RxTxRAMBuffer)
  ' init dma channel 2 for
  ' ECAN peripheral to dma transfer
  ECAN1Initialize(1, 3, 3, 3, 1, Can_Init_Flags)   ' initialize ECAN
  ECAN1SetBufferSize(ECAN1RAMBUFFERSIZE)
  ' set number of rx+tx buffers in DMA RAM

  ECAN1SelectTxBuffers(0x000F)
  ' select transmit buffers
  ' 0x000F = buffers 0:3 are transmit buffers
  ECAN1SetOperationMode(ECAN_MODE_CONFIG,0xFF)
  ' set CONFIGURATION mode


' continues..
```

```
' continued..

  ECAN1SetMask(ECAN_MASK_0, -1, ECAN_CONFIG_MATCH_MSG_TYPE and
ECAN_CONFIG_XTD_MSG)      ' set all mask1 bits to ones
  ECAN1SetMask(ECAN_MASK_1, -1, ECAN_CONFIG_MATCH_MSG_TYPE and
ECAN_CONFIG_XTD_MSG)      ' set all mask2 bits to ones
  ECAN1SetMask(ECAN_MASK_2, -1, ECAN_CONFIG_MATCH_MSG_TYPE and
ECAN_CONFIG_XTD_MSG)      ' set all mask3 bits to ones
  ECAN1SetFilter(ECAN_FILTER_10, 3, ECAN_MASK_2, ECAN_RX_BUFFER_7,
ECAN_CONFIG_XTD_MSG) ' set id of filter10  to 3,

' assign mask2 to filter10
' assign buffer7 to filter10
  ECAN1SetOperationMode(ECAN_MODE_NORMAL, 0xFF)' set NORMAL mode

  Tx_ID = 12111                                ' set transmit ID

  ECAN1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags)
  ' send initial message

  while TRUE  ' endless loop
    Msg_Rcvd = ECAN1Read(Rx_ID , RxTx_Data , Rx_Data_Len,
Can_Rcv_Flags) ' receive message
    if ((Rx_ID = 3) and (Msg_Rcvd <> 0)) <> 0 then
    ' if message received check id
      PORTB = RxTx_Data[ 0]
    ' id correct, output data at PORTB
      Inc(RxTx_Data[ 0] )
      Delay_ms(10)
      ECAN1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags)
    ' send incremented data back
    end if
  wend
end.
```


CAN RX of MCU
CAN TX of MCU
10R
1
2
3
4
8
7
6
5
TX-CAN RS
GND CANH
VCC CANL
RXD Vref
VCC
MCP2551
Shielded
twisted pair

EEPROM Library

EEPROM data memory is available with a number of dsPIC30 family. mikroBasic for dsPIC30/33 and PIC24 includes a library for comfortable work with MCU's internal EEPROM.

Note: Only dsPIC30 MCUs have EEPROM.

Library Routines

```
Eeprom_Erase
Eeprom_Erase_Block
Eeprom_Read
Eeprom_Write
Eeprom_Write_Block
```

Eeprom_Erase

Prototype	`sub procedure Eeprom_Erase(dim address as longint)`
Description	Erases a single (16-bit) location from EEPROM. Parameters : - `address`: address of the EEPROM location to be erased. **Note**: CPU is not halted for the Data Erase cycle. The user can poll WR bit, use NVMIF or Timer IRQ to detect the end of erase sequence.

Example	`dim eeAddr as longint`
	`...`
	`eeAddr = 0x7FFC80`
	`Eeprom_Erase(eeAddr)`

Eeprom_Erase_Block

Prototype	`sub procedure Eeprom_Erase_Block(dim address as longint)`
Returns	Nothing.
Description	Erases one EEPROM row (16 words block) from EEPROM. Parameters : - `address`: starting address of the EEPROM block to be erased. **Note**: CPU is not halted for the Data Erase cycle. The user can poll WR bit, use NVMIF or Timer IRQ to detect the end of erase sequence.

Example	`dim eeAddr as longint`
	`...`
	`eeAddr = 0x7FFC20`
	`Eeprom_Erase_Block(eeAddr)`

Eeprom_Read

Prototype	`sub function Eeprom_Read(dim address as longint) as word`
Returns	Word from the specified address.
Description	Reads data from specified address. Parameters : - `address`: address of the EEPROM location to be read.
Requires	It is the user's responsibility to obtain proper address parity (in this case, even).

Example	`dim eeAddr as longint`
	`  temp as word`
	`...`
	`eeAddr = 0x7FFC20`
	`temp = Eeprom_Read(eeAddr)`

Eeprom_Write

Prototype	`sub procedure Eeprom_Write(dim address as longint, dim data as word)`
Description	Writes data to specified address. Parameters : - `address`: address of the EEPROM location to be written. - `data`: data to be written. **Note**: Specified memory location will be erased before writing starts.

Example	`dim wrAddr as longint`
	`  eeData as word`
	`...`
	`eeData = 0xAAAA`
	`wrAddr = 0x7FFC30`
	`Eeprom_Write(wrAddr, eeData)`

Eeprom_Write_Block

Prototype	`sub procedure Eeprom_Write_Block(dim address as longint, dim byref data as word[100])`
Description	Writes one EEPROM row (16 words block) of data. Parameters : - `address`: starting address of the EEPROM memory block to be written. - `data`: data block to be written. **Note**: Specified memory block will be erased before writing starts.
Requires	It is the user's responsibility to maintain proper address alignment. In this case, address has to be a multiply of 32, which is the size (in bytes) of one row of MCU's EEPROM memory.

Example	`dim  wrAddr as longint`
	`  data as string[16]`
	`...`
	`wrAddr = 0x7FFC20`
	`data = "mikroElektronika"`
	`Eeprom_Write_Block(wrAddr, data)`

Library Example

The example demonstrates using EEPROM Library. Hardware configurations in this example are made for the dsPICPRO2 board and dsPIC30F6014A.

```
program EEPROM

const    iArr as word[16] = ("a", "b", "c", "d", "e", "f", "g",
"h", "i", "j", "k", "l", "m", "n", "o", 0)

dim eeRead as word
    eeAddr as longint

    dArr as string[16]
    iArr1 as word[16]
    i     as word

main:
  iArr1 = iArr
  '--- disable analog inputs
  ADPCFG = 0xFFFF

  TRISB  = 0
  LATB   = 0xFFFF
  eeRead = 0xAAAA
  eeAddr = 0x7FFC80
  while(eeRead < 0xAABA)
    Eeprom_Write(eeAddr, eeRead)
    eeRead = eeRead + 1
    while(NVMCON.15 <> 0)
      nop                          ' wait for write to finish,
    wend
    LATB = Eeprom_Read(eeAddr)     ' then, read the just-written
                                   ' data.
    eeAddr = eeAddr + 2
    Delay_ms(500)
  wend
  '--- write entire row of EEPROM data,...
  Eeprom_Write_Block(0x7FFC20, iArr1)
  while(NVMCON.15 <> 0)
    nop
  wend
  '--- ...read the data back...
  eeRead = 1
  eeAddr = 0x7FFC20
  i      = 0

  'continues...
```

```
  'continued...

while(eeRead <> 0)
    eeRead   = Eeprom_Read(eeAddr)
    dArr[ i]  = Lo(eeRead)
    i         = i + 1
    eeAddr   = eeAddr + 2
  wend
  '--- ...and display it on LCD.
  Lcd_Init_DsPicPro2()
  Lcd_Out(2,1, dArr)
end.
```

Flash Memory Library

This library provides routines for accessing microcontroller's (internal) Flash memory.

On the dsPIC30/33 and PIC24, Flash memory is mapped to address space 3:2, which means that every 3 consecutive bytes of Flash have 2 consecutive address locations available. That is why mikroE's library allows data to be written to flash in two ways: "regular" and "compact". In the "regular" mode, which is used for word(16-bit) variables, the 3rd (un-addressable) flash memory byte remains unused. In the "compact" mode, which can be used for 1 byte-sized variables/arrays, all flash bytes are being used.

All dsPIC30/33 and PIC24 MCUs use the RTSP module to perform Read/Erase/Write operations on Flash memory. This, together with the internal structure of the Flash, imposes certain rules to be followed when working with Flash memory:

dsPIC30:

- Erasing can be done only in 32-instructions (64 addresses, 96 bytes) memory blocks. This means that the block start address should be a multiply of 64 (i.e. have 6 lower bits set to zero).
- Data is read and written in 4-instructions (8 addresses, 12 bytes) blocks.This means that the block start address should be a multiply of 8 (i.e. have 3 lower bits set to zero).
- On the dsPIC30s, 2 address locations are assigned on every 3 bytes of (flash) program memory. Due to this specific and non-one-to-one address mapping, the mikroBasic for dsPIC30/33 and PIC24 offers two sets of Flash handling functions: "regular" and "compact".
  Using the "regular" set, the user can write one byte of data to a single address, which means that each byte of written data has its own address, but on every 2 written bytes one byte of Flash memory remains empty.
  Using the "compact" set, every byte of Flash memory, including those non-addressable, is filled with data; this method can only be used for data organized in bytes. The "compact" functions have `_Compact` as name suffix.
- For run-time FLASH read/write, the dsPIC30's RTSP module is being used. It organizes data into rows and panels. Each row contains write latches that can hold 4 instructions (12 bytes). The number of panels varies from one dsPIC30 MCU model to another. Because of that, the flash write sequence has been split into several operations (`_Write_Init()`, `_Write_LoadLatch4()`, `_Write_DoWrite()`), in order to be usable on all dsPICs.

PIC24 and dsPIC33:

Erasing can be done only in 512-instructions (1024 addresses, 1536 bytes) memory blocks, which means that the block start address should be a multiply of 1024 (i.e. have 10 lower bits set to zero).
Data is read and written in 64-instructions (128 addresses, 192 bytes) blocks.This means that the block start address should be a multiply of 128 (i.e. have 7 lower bits set to zero).
On the dsPIC33 and PIC24s, 2 address locations are assigned on every 3 bytes of (flash) program memory. Due to this specific and non-one-to-one address mapping, the mikroBasic offers two sets of Flash handling functions: "regular" and "compact".
Using the "regular" set, the user can write one byte of data to a single address, which means that each byte of written data has its own address, but on every 2 written bytes one byte of Flash memory remains empty.
Using the "compact" set, every byte of Flash memory, including those non-addressable, is filled with data; this method can only be used for data organized in bytes.
The "compact" functions have _Compact as name suffix.

Library Routines

dsPIC30:

```
Flash_Erase32

Flash_Write_Block
Flash_Write_Compact
Flash_Write_Init
Flash_Write_Loadlatch4
Flash_Write_Loadlatch4_Compact
Flash_Write_DoWrite

Flash_Read4
Flash_Read4_Compact
```

PIC24 and dsPIC33:

```
Flash_Erase
Flash_Write
Flash_Write_Compact
Flash_Read
Flash_Read_Compact
```

dsPIC30:

Flash_Erase32

Prototype	`sub procedure Flash_Erase32(dim flash_address as longint)`
Description	Erases one block (32 instructions, 64 addresses, 96 bytes) from the program FLASH memory. Parameters : - `flash_address`: starting address of the FLASH memory block
Requires	**Note**: The user should take care about the address alignment (see the explanation at the beginning of this page).
Example	`' erase the 32-instruction block, starting from address 0x006000` `Flash_Erase32(0x006000)`

Flash_Write_Block

Prototype	`sub procedure Flash_Write_Block(dim flash_address as longint, dim data_address as word)`
Description	Fills one writeable block of Flash memory (4 instructions, 8 addresses, 12 bytes) in the "regular" mode. Addresses and data are being mapped 1-on-1. This also means that the 3rd byte of each program location remains unused. Parameters : - `flash_address`: starting address of the FLASH memory block - `data_address`: starting address of the data block to be written
Requires	The block to be written to must be erased first, either from the user code (through the RTSP), or during the programming of MCU. Please note that block size that is to be erased is different from that one that can be written with this function! **Note**: The user should take care about the address alignment (see the explanation at the beginning of this page).

Example	`dim flash_address as longint`
	`  cArr  as string[ 4]`
	`  ptr_data  as word`
	`...`
	`flash_address = 0x006000`
	`cArr = "ABCD"`
	`ptr_data = @cArr`
	`Flash_Write_Block(flash_address, ptr_data)`

Flash_Write_Compact

Prototype	`sub procedure Flash_Write_Compact(dim flash_address as longint, dim data_address as word, dim bytes as word)`
Description	Fills a portion of Flash memory using the dsPIC30 RTSP module, in the "compact" manner. In this way, several blocks of RTSP's latch can be written in one pass. One latch block contains 4 instructions (8 addresses, 12 bytes). Up to 8 latch blocks can be written in one round, resulting in a total of 8*12 = 96 bytes. This method uses all available bytes of the program FLASH memory, including those that are not mapped to address space (every 3rd byte). Parameters : - `flash_address`: starting address of the FLASH memory block - `data_address`: starting address of the data block to be written - `bytes`: number of bytes to be written. The amount of bytes to be written must be a multiply of 12, since this is the size of the RTSP's write latch(es).
Requires	The block to be written to must be erased first, either from the user code Flash_Erase32, or during the programming of MCU. Please note that block size that is to be erased is different from the one that can be written with this function! **Note**: The user should take care about the address alignment (see the explanation at the beginning of this page).

Example	`dim flash_address as longint`
	`  cArr  as string[ 36]`
	`  ptr_data  as word`
	`...`
	`flash_address = 0x006000`
	`cArr = "mikroElektronika12mikroElektronika34"`
	`ptr_data = @cArr`
	`Flash_Write_Compact(flash_address, ptr_data, 36)`

Flash_Write_Init

Prototype	`sub procedure Flash_Write_Init(dim flash_address as longint, dim data_address as word)`
Description	Initializes RTSP for write-to-FLASH operation. Parameters : - `flash_address`: starting address of the FLASH memory block - `data_address`: starting address of the data block to be written
Requires	The block to be written to must be erased first, either from the user code Flash_Erase32, or during the programming of MCU. Please note that block size that is to be erased is different from that one that can be written with this function! **Note**: The user should take care about the address alignment (see the explanation at the beginning of this page).

Example	`const iArr as word[ 8] = ("m", "i", "k", "r", "o", "E", "l", "e")`
	`dim ptr_data as word`
	`  ...`
	`  ptr_data = @iArr`
	`  Flash_Write_Init(0x006100, ptr_data)`
	`  Flash_Write_Loadlatch4()`
	`  Flash_Write_Loadlatch4()`
	`  Flash_Write_DoWrite()`

Flash_Write_Loadlatch4

Prototype	`sub procedure Flash_Write_Loadlatch4()`
Description	Loads the current RTSP write latch with data (4 instructions, 8 addresses, 12 bytes). The data is filled in the "regular" mode.
Requires	The block to be written to must be erased first, either from the user code Flash_Erase32, or during the programming of MCU. Please note that block size that is to be erased is different from that one that can be written with this function! This function is used as a part of the Flash write sequence, therefore the Flash_Write_Init function must be called before this one. This function can be called several times before commiting the actual write-to-Flash operation Flash_Write_DoWrite. This depends on the organization of the RTSP module for the certain dsPIC30. Please consult the Datasheet for particular dsPIC30 on this subject.

Example	`const iArr as word[8] = ("m", "i", "k", "r", "o", "E", "l", "e")`
	`dim ptr_data as word`
	`...`
	`ptr_data = @iArr`
	`Flash_Write_Init(0x006100, ptr_data)`
	`Flash_Write_Loadlatch4()`
	`Flash_Write_Loadlatch4()`
	`Flash_Write_DoWrite()`

Flash_Write_Loadlatch4_Compact

Prototype	`sub procedure Flash_Write_Loadlatch4_Compact()`
Description	Loads the current RTSP write latch with data (4 instructions, 8 addresses, 12 bytes). The data is filled in the "compact" mode.
Requires	The block to be written to must be erased first, either from the user code Flash_Erase32, or during the programming of MCU. Please note that block size that is to be erased is different from that one that can be written with this function! This function is used as a part of the Flash write sequence, therefore the Flash_Write_Init function must be called before this one. This function can be called several times before committing actual write-to-Flash operation Flash_Write_DoWrite. This depends on the organization of the RTSP module for the certain dsPIC30. Please consult the Datasheet for particular dsPIC30 on this subject.

Example	`const iArr as word[8] = ("m", "i", "k", "r", "o", "E", "l", "e")`
	`dim ptr_data as word`
	`...`
	`ptr_data = @iArr`
	`Flash_Write_Init(0x006100, ptr_data)`
	`Flash_Write_Loadlatch4_Compact()`
	`Flash_Write_Loadlatch4_Compact()`
	`Flash_Write_DoWrite()`

Flash_Write_DoWrite

Prototype	`sub procedure Flash_Write_DoWrite()`
Description	Commits the FLASH write operation.
Requires	The block to be written to must be erased first, either from the user code Flash_Erase32, or during the programming of MCU. Please note that block size that is to be erased is different from that one that can be written with this function! This function is used as a part of the Flash write sequence, therefore Flash_Write_Init and certain number of Flash_Write_Loadlatch4 or Flash_Write_Loadlatch4_Compact function calls must be made before this one. This function is to be called once, at the end of the FLASH write sequence.

Example	`const iArr as word[8] = ("m", "i", "k", "r", "o", "E", "l", "e")`
	`dim ptr_data as word`
	`  ...`
	`  ptr_data = @iArr`
	`  Flash_Write_Init(0x006100, ptr_data)`
	`  Flash_Write_Loadlatch4()`
	`  Flash_Write_Loadlatch4()`
	`  Flash_Write_DoWrite()`

Flash_Read4

Prototype	`sub procedure Flash_Read4(dim flash_address as longint, dim write_to as word)`
Description	Reads one latch row (4 instructions, 8 addresses) in the "regular" mode. Parameters : - `flash_address` starting address of the FLASH memory block to be read - `write_to`: starting address of RAM buffer for storing read data
Requires	**Note**: The user should take care of the address alignment (see the explanation at the beginning of this page).

Example	`dim flash_address as longint`
	`  cArr  as word[ 4]`
	`  ptr_data  as word`
	`...`
	`flash_address = 0x006000`
	`ptr_data = @cArr`
	`Flash_Read4(flash_address, ptr_data)`

Flash_Read4_Compact

Prototype	`sub procedure Flash_Read4_Compact(dim flash_address as longint, dim write_to as word)`
Description	Reads one latch row (4 instructions, 8 addresses) in the "compact" mode. Parameters : - `flash_address`: starting address of the FLASH memory block to be read - `write_to`: starting address of RAM buffer for storing read data
Requires	**Note**: The user should take care of the address alignment (see the explanation at the beginning of this page).

Example	`dim flash_address as longint`
	`  cArr  as word[ 8]`
	`  ptr_data  as word`
	`...`
	`flash_address = 0x006000`
	`ptr_data = @cArr`
	`Flash_Read4_Compact(flash_address, ptr_data)`

PIC24 and dsPIC33:

Flash_Erase

Prototype	`sub procedure Flash_Erase(dim address as longint)`
Description	Erases one block (512 instructions, 1024 addresses, 1536 bytes) from the program FLASH memory. Parameters : - `address`: starting address of the FLASH memory block
Requires	**Note**: The user should take care about the address alignment (see the explanation at the beginning of this page).

Example	`'--- erase the flash memory block, starting from address 0x006400`
	`dim flash_address as longint`
	`...`
	`flash_address = 0x006400`
	`Flash_Erase(flash_address)`

Flash_Write

Prototype	`sub procedure Flash_Write(dim address as longint, dim byref data as word[ 64] )`
Description	Fills one writeable block of Flash memory (64 instructions, 128 addresses, 192 bytes) in the "regular" mode. Addresses and data are being mapped 1-on-1. This also means that the 3rd byte of each program location remains unused. Parameters : - `address`: starting address of the FLASH memory block - `data`: data to be written
Requires	The block to be written to must be erased first, either from the user code (through the RTSP), or during the programming of MCU. Please note that block size that is to be erased is different from that one that can be written with this function! **Note**: The user should take care about the address alignment (see the explanation at the beginning of this page).

Example	`dim data as word[ 64] = { "m", "i", "k", "r", "o", "E", "l", "e", "k", "t", "r", "o", "n", "i", "k", "a"}`
	`...`
	
	`Flash_Write(0x006500, data)`

Flash_Write_Compact

Prototype	`sub procedure Flash_Write_Compact(dim address as longint, dim byref data as byte[ 192] )`
Description	Fills a portion of Flash memory (64 instructions, 128 addresses, 192 bytes) using the dsPIC33 and PIC24s RTSP (Run Time Self Programming) module, in the "compact" manner. This method uses all available bytes of the program FLASH memory, including those that are not mapped to address space (every 3rd byte). Parameters : - `address`: starting address of the FLASH memory block - `data`: data to be written
Requires	The block to be written to must be erased first, either from the user code (Flash_Erase), or during the programming of MCU. Please note that block size that is to be erased is different from that one that can be written with this function! **Note**: The user should take care of the address alignment (see the explanation at the beginning of this page).

Example	`dim data as string[ 192]`
	`...`
	`data = "supercalifragillisticexpialidotiousABCDEFGHIJKLMNOPRSTUVWXYZ1234"`
	
	`Flash_Write_Compact(0x006400, data)`

Flash_Read

Prototype	`sub procedure Flash_Read(dim address as longint, dim byref write_to as word[ 100] , dim NoWords as word)`
Returns	Address of RAM buffer for storing read data.
Description	Reads required number of words from the flash memory in the "regular" mode. Parameters : address: starting address of the FLASH memory block to be read write_to: starting address of RAM buffer for storing read data NoWords: number of words to be read
Requires	**Note**: The user should take care of the address alignment (see the explanation at the beginning of this page).

Example	`dim Buffer as word[ 10]`
	`  start_address as longint`
	`...`
	`Flash_Write(0x006500, data)`
	`start_address = 0x6500`
	`Flash_Read(start_address, Buffer, 10)`

Flash_Read_Compact

Prototype	`sub procedure Flash_Read_Compact(dim address as longint, dim byref write_to as byte[ 100] , dim NoBytes as word)`
Returns	Address of RAM buffer for storing read data.
Description	Reads required number of bytes from the flash memory in the "compact" mode. Parameters : - `address`: starting address of the FLASH memory block to be read - `write_to`: starting address of RAM buffer for storing read data - `NoBytes`: number of bytes to be read
Requires	**Note**: The user should take care of the address alignment (see the explanation at the beginning of this page).

Example	`dim Buffer as byte[ 10]`
	`  start_address as longint`
	`...`
	`Flash_Write(0x006500, data)`
	`start_address = 0x6500`
	`Flash_Read(start_address, Buffer, 10)`

Library Example

In these examples, various techniques of reading/writing from/to on-chip FLASH memory are shown.

dsPIC30:

```
program Flash_Test

const  iArr as word[ 8] = ("m", "i", "k", "r", "o", "E", "l", "e")

dim  cArr as string[ 35]
     cArr2 as byte[ 40]
     jhh as word[ 16]
     pv1 as word
     bb as word
     i  as word

main:
  ADPCFG = 0xFFFF
  cArr = "mikroElektronikaYmikroElektronikaXX"
  pv1 = @cArr

  '--- erase the block first
  Flash_Erase32(0x006000)
  '--- write compact format to flash
  Flash_Write_Compact(0x006000, pv1, 36)
  '--- read compact format
  pv1 = @cArr2
  Flash_Read4_Compact(0x006000, pv1)
  pv1 = pv1 + 12
  Flash_Read4_Compact(0x006008, pv1)
  pv1 = pv1 + 12
  Flash_Read4_Compact(0x006010, pv1)
  pv1 = pv1 + 12

  '--- show what has been written
  i = 0
  Uart1_Init(9600)
  Delay_ms(100)
  Uart1_Write_Char("s") Uart1_Write_Char("t") Uart1_Write_Char("a")
  Uart1_Write_Char("r")
  Uart1_Write_Char("t") Uart1_Write_Char(10)



  ' continues...
```

```
    ' continued...

    while i <> 35
      bb = cArr2[ i]
      i = i + 1
      Uart1_Write_Char(bb)
    wend

    '--- now for some non-compact flash-write
    jhh = iArr
    pv1 = @jhh
    Flash_Write_Init(0x006100, pv1)
    Flash_Write_Loadlatch4()
    Flash_Write_Loadlatch4()
    Flash_Write_DoWrite()
  end.
```

PIC24 and dsPIC33:

In this example, various read/write tecniques to/from the on-chip FLASH memory are shown.

```
program FLASH_Test

const iArr  as word[ 64] = ("m", "i", "k", "r", "o", "E", "l", "e", "k", "t",
"r", "o", "n", "i", "k", "a")
dim   cArr  as string[ 192]
      cArr2  as word[ 65]
       i as word

main:
  AD1PCFGH = $FFFF
  AD1PCFGL = $FFFF

  PORTB    = 0x0004
  TRISB    = 1
  cArr     = "supercalifragillisticexpialidotiousABCDEFGHIJKLMNOPRSTUVWXYZ1234"

  ' press RB0 to continue
  while (PORTB.0 = 0)
    asm
      nop
    end asm
  wend

  '--- erase the block first
  Flash_Erase(0x006400)

  'continues...
```

```
  'continued...

  cArr2 = iArr

  for i=16 to 63
    cArr2[ i] = "x"
  next i

  '--- now for some non-compact flash-write
  Flash_Write(0x006500, cArr2)

  PORTB = 0x0008

  '--- write compact format to flash
  for i=64 to 191
    cArr[ i] = "y"
  next i

  Flash_Write_Compact(0x006400, cArr)

  PORTB = 0x0010

  '--- read compact format
  Flash_Read_Compact(0x006400, cArr, 64)
  cArr[ 64] = 0  'termination

  '--- show what has been written
  i = 0
  Uart1_Init(9600)
  Uart1_Write_Text("start : ")
  Uart1_Write_Char(10)
  while(cArr[ i] <> 0)
    Uart1_Write_Char(cArr[ i] )
    i = i + 1
  wend

  PORTB = 0x0018
end.
```

GLCD Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for drawing and writing on the commonly used Samsung (KS108/KS107) Graphic LCD 128x64.

For creating a custom set of GLCD images use GLCD Bitmap Editor Tool.

Note: mikroElektronika's development system based initialization routines can be found in the setup library files located in the Uses folder.

Library Routines

Basic routines:

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

Advanced routines:

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

Glcd_Init

Prototype	`sub procedure Glcd_Init(dim byref cs1_port as word, dim cs1_pin as word, dim byref cs2_port as word, dim cs2_pin as word, dim byref rs_port as word, dim rs_pin as word, dim byref rw_port as word, dim rw_pin as word, dim byref rst_port as word, dim rst_pin as word, dim byref en_port as word, dim en_pin as word, dim byref data_port as word)`
Description	Initializes the GLCD module. Each of the control lines is both port and pin configurable, while data lines must be on a single port (pins <0:7>). Parameters : - `cs1_port`: **chip select 1** signal port address - `cs1_pin`: **chip select 1** signal pin - `cs2_port`: **chip select 2** signal port address - `cs2_pin`: **chip select 2** signal pin - `rs_port`: **register select** (data/instruction) signal port address - `rs_pin`: **register select** (data/instruction) signal pin - `rw_port`: **read/write** signal port address - `rw_pin`: **read/write** signal pin - `rst_port`: **reset** signal port address - `rst_pin`: **reset** signal pin - `en_port`: **enable** signal port address - `en_pin`: **enable** signal pin - `data_port`: **data** port
Example	`' Init for EasydsPIC2 development system` `Glcd_Init(PORTF,0,PORTF,1,PORTD,0,PORTD,1,PORTD,3,PORTD,2,PORTB)`

Glcd_Config

Prototype	`sub procedure Glcd_Config(dim byref cs1_port as word, dim cs1_pin as word, dim byref cs2_port as word, dim cs2_pin as word, dim byref rs_port as word, dim rs_pin as word, dim byref rw_port as word, dim rw_pin as word, dim byref rst_port as word, dim rst_pin as word, dim byref en_port as word, dim en_pin as word, dim byref data_port_Lo as word, dim byref data_port_Hi as word, dim ports_config as byte)`
Description	Initializes the GLCD module. Each of the control lines is both port and pin configurable. GLCD data lines can be configured on two MCU ports. Parameters : - `cs1_port`: **chip select 1** signal port address - `cs1_pin`: **chip select 1** signal pin - `cs2_port`: **chip select 2** signal port address - `cs2_pin`: **chip select 2** signal pin - `rs_port`: **register select** (data/instruction) signal port address - `rs_pin`: **register select** (data/instruction) signal pin - `rw_port`: **read/write** signal port address - `rw_pin`: **read/write** signal pin - `rst_port`: **reset** signal port address - `rst_pin`: **reset** signal pin - `en_port`: **enable** signal port address - `en_pin`: **enable** signal pin - `data_port_Lo`: MCU's port connected to GLCD's D3..D0 data lines - `data_port_Hi`: MCU's port connected to GLCD's D7..D4 data lines - `ports_config`: connection descriptor. Valid values: 0..3. The `ports_config` values determine which nibble of data_port_Lo and data_port_Hi is used: 0 = data_port_Lo pins<3..0> and data_port_Hi pins<3..0> 1 = data_port_Lo pins<3..0> and data_port_Hi pins<7..4> 2 = data_port_Lo pins<7..4> and data_port_Hi pins<3..0> 3 = data_port_Lo pins<7..4> and data_port_Hi pins<7..4>

Example	`' Init for EasydsPIC3 development system`
	`Glcd_Config(PORTB,4,PORTB,5,PORTF,0,PORTF, 1,PORTF, 5,PORTF, 4,PORTB,PORTD, 0)`
	
	`' or just call the appropriate system based`
	`' initialization routine`
	`Glcd_Init_EasyDsPIC3()`

Glcd_Set_Side

Prototype	`sub procedure Glcd_Set_Side(dim x_pos as word)`
Description	Selects GLCD side. Refer to the GLCD datasheet for detailed explaination. Parameters : - `x_pos`: position on x-axis. Valid values: 0..127 The parameter x_pos specifies the GLCD side: values from 0 to 63 specify the left side, values from 64 to 127 specify the right side. **Note**: For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines.

Example	The following two lines are equivalent, and both of them select the left side of GLCD:
	
	`Glcd_Select_Side(0)`
	`Glcd_Select_Side(10)`

Glcd_Set_X

Prototype	`sub procedure Glcd_Set_X(dim x_pos as word)`
Description	Sets x-axis position to x_pos dots from the left border of GLCD within the selected side. Parameters : - `x_pos`: position on x-axis. Valid values: 0..63 **Note**: For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines.
Example	`Glcd_Set_X(25)`

Glcd_Set_Page

Prototype	`sub procedure Glcd_Set_Page(dim page as word)`
Description	Selects page of the GLCD. Parameters : - `page`: page number. Valid values: 0..7 Note: For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines.
Example	`Glcd_Set_Page(5)`

Glcd_Read_Data

Prototype	`sub function Glcd_Read_Data() as word`
Returns	One byte from GLCD memory, formatted as a word (16-bit).
Description	Reads data from the current location of GLCD memory and moves to the next location.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines. GLCD side, x-axis position and page should be set first. See functions `Glcd_Set_Side`, `Glcd_Set_X`, and `Glcd_Set_Page`.

Example	`dim data as word`
	`...`
	`data = Glcd_Read_Data()`

Glcd_Write_Data

Prototype	`sub procedure Glcd_Write_Data(dim ddata as word)`
Description	Writes one byte to the current location in GLCD memory and moves to the next location. Parameters : - `ddata`: data to be written
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines. GLCD side, x-axis position and page should be set first. See functions `Glcd_Set_Side`, `Glcd_Set_X`, and `Glcd_Set_Page`.

Example	`dim data as byte`
	`...`
	`Glcd_Write_Data(data)`

Glcd_Fill

Prototype	`sub procedure Glcd_Fill(dim pattern as word)`
Description	Fills GLCD memory with the byte pattern. Parameters : - `pattern`: byte to fill GLCD memory with To clear the GLCD screen, use `Glcd_Fill(0)`. To fill the screen completely, use `Glcd_Fill(0xFF)`.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines.
Example	`Glcd_Fill(0)  ' Clear screen`

Glcd_Dot

Prototype	`sub procedure Glcd_Dot(dim x_pos, y_pos, color as word)`
Description	Draws a dot on GLCD at coordinates (x_pos, y_pos). Parameters : - `x_pos`: x position. Valid values: 0..127 - `y_pos`: y position. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines a dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state. **Note**: For x and y axis layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines.
Example	`' Invert the dot in the upper left corner` `Glcd_Dot(0, 0, 2)`

Glcd_Line

Prototype	`sub procedure Glcd_Line(dim x_start, y_start, x_end, y_end as integer, dim color as word)`
Description	Draws a line on GLCD. Parameters : - `x_start`: x coordinate of the line start. Valid values: 0..127 - `y_start`: y coordinate of the line start. Valid values: 0..63 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized, see `Glcd_Init` and `Glcd_Config` routines.
Example	`' Draw a line between dots (0,0) and (20,30)` `Glcd_Line(0, 0, 20, 30, 1)`

Glcd_V_Line

Prototype	`sub procedure Glcd_V_Line(dim y_start, y_end, x_pos, color as word)`
Description	Draws a vertical line on GLCD. Parameters : - `y_start`: y coordinate of the line start. Valid values: 0..63 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `x_pos`: x coordinate of vertical line. Valid values: 0..127 - `color`: color parameter. Valid values: 0..2 The parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized, see `Glcd_Init` and `Glcd_Config` routines.
Example	`' Draw a vertical line between dots (10,5) and (10,25)` `Glcd_V_Line(5, 25, 10, 1)`

Glcd_H_Line

Prototype	`sub procedure Glcd_H_Line(dim x_start, x_end, y_pos, color as word)`
Description	Draws a horizontal line on GLCD. Parameters : - `x_start`: x coordinate of the line start. Valid values: 0..127 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_pos`: y coordinate of horizontal line. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized, see `Glcd_Init` and `Glcd_Config` routines.
Example	`' Draw a horizontal line between dots (10,20) and (50,20)` `Glcd_H_Line(10, 50, 20, 1)`

Glcd_Rectangle

Prototype	`sub procedure Glcd_Rectangle(dim x_upper_left, y_upper_left, x_bottom_right, y_bottom_right, color as word)`
Description	Draws a rectangle on GLCD. Parameters : - `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`:x coordinate of the lower right rectangle corner.Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines.
Example	`' Draw a rectangle between dots (5,5) and (40,40)` `Glcd_Rectangle(5, 5, 40, 40, 1)`

Glcd_Box

Prototype	`sub procedure Glcd_Box(dim x_upper_left, y_upper_left, x_bottom_right, y_bottom_right, color as word)`
Description	Draws a box on GLCD. Parameters : - `x_upper_left`: x coordinate of the upper left box corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left box corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right box corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right box corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the box fill: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines.
Example	`' Draw a box between dots (5,15) and (20,40)` `Glcd_Box(5, 15, 20, 40, 1)`

Glcd_Circle

Prototype	`sub procedure Glcd_Circle(dim x_center, y_center, radius as integer, dim color as word)`
Description	Draws a circle on GLCD. Parameters : - `x_center`: x coordinate of the circle center. Valid values: 0..127 - `y_center`: y coordinate of the circle center. Valid values: 0..63 - `radius`: radius size - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the circle line: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines.
Example	`' Draw a circle with center in (50,50) and radius=10` `Glcd_Circle(50, 50, 10, 1)`

Glcd_Set_Font

Prototype	`sub procedure Glcd_Set_Font(dim byref const activeFont as byte, dim aFontWidth, aFontHeight as word, dim aFontOffs as word)`
Description	Sets font that will be used with the Glcd_Write_Char and Glcd_Write_Text routines. Parameters : - `activeFont`: font to be set. Needs to be formatted as an array of char - `aFontWidth`: width of the font characters in dots. - `aFontHeight`: height of the font characters in dots. - `aFontOffs`: number that represents difference between mikroBasic for dsPIC30/33 and PIC24 character set and regular ASCII set (eg. if 'A' is 65 in ASCII character, and 'A' is 45 in mikroBasic for dsPIC30/33 and PIC24 character set, aFontOffs is 20). Demo fonts supplied with the library have an offset of 32, which means that they start with space. The user can use fonts given in the file "__Lib_Glcd_Fonts.dbas" file located in the Uses\Consts folder or create his own fonts.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines.
Example	`' Use the custom 5x7 font "myfont" which starts with space (32):` `Glcd_Set_Font(@myfont, 5, 7, 32)`

Glcd_Write_Char

Prototype	`sub procedure Glcd_Write_Char(dim ch, x_pos, page_num, color as word)`
Description	Prints character on the GLCD. Parameters : - `ch`: character to be written - `x_pos`: character starting position on x-axis. Valid values: 0..(127-FontWidth) - `page_num`: the number of the page on which character will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the character: 0 white, 1 black, and 2 inverts each dot. **Note**: For x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines. Use `Glcd_Set_Font` to specify the font for display; if no font is specified, then default 5x8 font supplied with the library will be used.
Example	`' Write character 'C' on the position 10 inside the page 2:` `Glcd_Write_Char('C', 10, 2, 1)`

Glcd_Write_Text

Prototype	`sub procedure Glcd_Write_Text(dim byref text as char[40], dim x_pos, page_num, color as word)`
Description	Prints text on GLCD. Parameters : - `text`: text to be written - `x_pos`: text starting position on x-axis. - `page_num`: the number of the page on which text will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the text: 0 white, 1 black, and 2 inverts each dot. **Note**: For x axis and page layout explanation, see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines. Use Glcd_Set_Font to specify the font for display; if no font is specified, then default 5x8 font supplied with the library will be used.
Example	`' Write text "Hello world!" on the position 10 inside the page 2:` `Glcd_Write_Text("Hello world!", 10, 2, 1)`

Glcd_Image

Prototype	`sub procedure Glcd_Image(dim byref constimage as byte)`
Description	Displays bitmap on GLCD. Parameters : - `image`: image to be displayed. Bitmap array can be located in both code and RAM (due to the mikroBasic for dsPIC30/33 and PIC24 pointer to const and pointer to RAM equivalency). Use the mikroBasic's integrated GLCD Bitmap Editor (menu option **Tools › GLCD Bitmap Editor**) to convert image to a constant array suitable for displaying on GLCD.
Requires	GLCD needs to be initialized, see the `Glcd_Init` and `Glcd_Config` routines.
Example	`' Draw image my_image on GLCD` `Glcd_Image(my_image)`

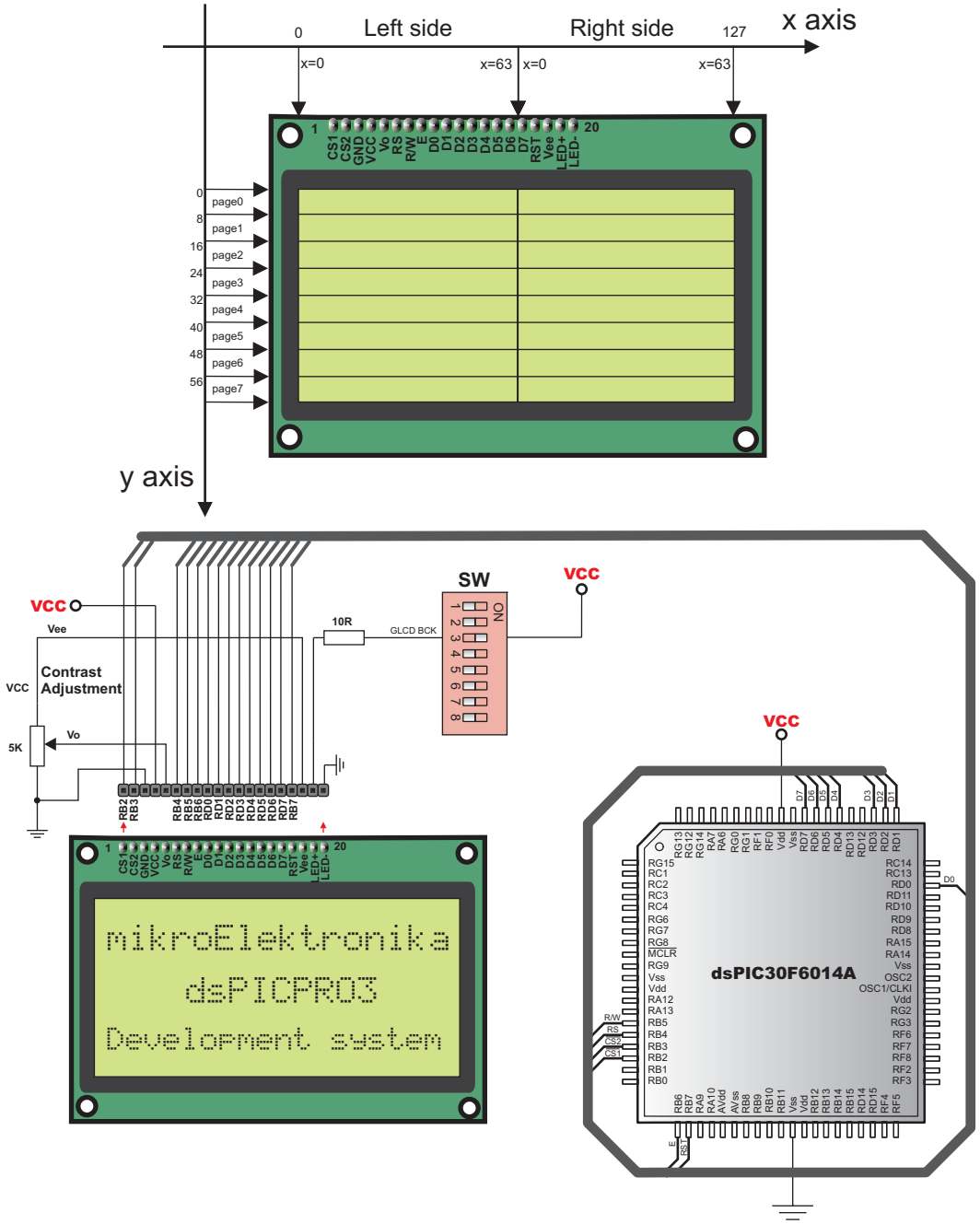
Library Example

The following example is a simple demonstration of the GLCD Library:

```
program GlcdDemo
include "__Lib_Glcd_Images"

dim ii as byte
    jj as word
    someText  as char[20]

sub procedure Delay2S
  Delay_ms(2000)
end sub

main:
  ADPCFG  = $FFFF
  TBLPAG  = 0

  '** dsPICPro3
  Glcd_Init_dsPICPro3
  'Glcd_Init(PORTB, 2, 3, PORTB, 4, 5, 7, 6, PORTD)

  Glcd_Fill(0xAA)
  Delay2S()

  ' Main loop
  while TRUE
    Glcd_Fill(0x00)   ' Clear screen

    ' Draw image
    Glcd_Image(truck_bmp)
    Delay2S()

    ' Draw dots
    for jj  = 0 to 40
      Glcd_Dot(jj, jj, 1)
    next jj

    Delay2S()

    ' Draw lines
    Glcd_Fill(0x00)
    Glcd_Line(120, 1, 5, 60, 1)
    Delay2S()
    Glcd_Line(12, 42, 5, 60, 1)

    Delay2S()
' continues...
```

```
' continued...

    Glcd_H_Line(5, 15, 6, 1)
    Glcd_V_Line(6, 15, 15, 1)

    ' Draw rectangle
    Glcd_Rectangle(12, 20, 93,57, 1)

    Delay2S()

    ' Draw lines
    Glcd_Line(120, 12, 12,60, 1)

    Delay2S()

    Glcd_H_Line(5, 15, 6, 1)
    Glcd_Line(0, 12, 120, 60, 1)
    Glcd_V_Line(7, 63, 127, 1)

    Delay2S()

    ' Draw circles
    for ii  = 1 to 10
      Glcd_Circle(63, 31, 3*ii, 1)
    next ii

    Delay2S()

    ' Draw box
    Glcd_Box(12, 20, 70, 57, 2)

    Delay2S()

    Glcd_Fill(0x00)

    ' Font demo
    Glcd_Set_Font(@System3x6, 3, 6, 32)
    someText = "SMALL FONT AS 3X6"
    Glcd_Write_Text(someText, 20, 5, 1)

    Glcd_Set_Font(@FontSystem5x8, 5, 8, 32)
    someText = "Large Font 5x8"
    Glcd_Write_Text(someText, 3, 4, 1)

    Delay2S()
  wend
end.
```

Hardware Connection



I2C Library

The I²C full master I²C module is available with a number of the dsPIC30/33 and PIC24 MCU models. mikroBasic for dsPIC30/33 and PIC24 provides a library which supports the master I²C mode.

Note: For the dsPIC33 and PIC24 MCUs with the multiple I²C modules there are I2C1 (supports I²C1 module), I2C2 (supports I²C2 module) and I2C libraries. I2C library supports I²C1 module and it is kept to maintain backwards compatibility with the previous versions of mikroBasic for dsPIC30/33 and PIC24.

Library Routines

dsPIC30:

```
I2C_Init
I2C_Start
I2C_Restart
I2C_Wait_For_Idle
I2C_Read
I2C_Write
I2C_Stop
```

PIC24 and dsPIC33:

```
I2C1_Init
I2C1_Start
I2C1_Restart
I2C1_Wait_For_Idle
I2C1_Read
I2C1_Write
I2C1_Stop
```

```
I2C2_Init
I2C2_Start
I2C2_Restart
I2C2_Wait_For_Idle
I2C2_Read
I2C2_Write
I2C2_Stop
```

```
I2C_Init
I2C_Start
I2C_Restart
I2C_Wait_For_Idle
I2C_Read
I2C_Write
I2C_Stop
```

I2C1_Init

Prototype	`sub procedure I2c1_Init(dim scl as longint)`
Description	Initializes the I²C1 module. This function enables the I2C1 module by setting the I2CEN bit. The rest of the bits in I2C1CON register remains unchanged. Default initialization (after reset) of the I2C1 module is: - continue operation in IDLE mode - IPMI mode disabled - 7-bit slave address - slew rate control enabled - general call address disabled - software or receive clock stretching disabled Parameters : - `scl`: requested serial clock rate. Refer to the MCU's datasheet for correct values of the scl in respect with *Fosc*.
Requires	MCU with the I²C1 module. Supported by the dsPIC33 and PIC24 MCUs only.
Example	`' Initialize the I2C1 module with clock_rate=100000` `I2C1_Init(100000)`

I2C1_Start

Prototype	`sub procedure I2c1_Start()`
Description	Determines if the I²C bus is free and issues START signal.
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C1_Init. Supported by the dsPIC33 and PIC24 MCUs only.
Example	`' Issue START signal` `I2C1_Start()`

I2C1_Restart

Prototype	`sub procedure I2C1_Restart()`
Description	Issues repeated START signal.
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C1_Init. Supported by the dsPIC33 and PIC24 MCUs only.
Example	`' Issue RESTART signal` `I2C1_Restart()`

I2C1_Wait_For_Idle

Prototype	`sub procedure I2C1_Wait_For_Idle()`
Description	Waits for the I²C bus to become free. This is a blocking function.
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C1_Init. Supported by the dsPIC33 and PIC24 MCUs only.

Example	`dim data as byte`
	`...`
	`I2C1_Wait_For_Idle()`
	`I2C1_Write(data)`
	`...`

I2C1_Read

Prototype	`sub function I2c1_Read(dim ack as word) as byte`
Returns	Received data.
Description	Reads a byte from the I²C bus. Parameters : - `ack`: acknowledge signal parameter. If the `ack==0` acknowledge signal will be sent after reading, otherwise the not acknowledge signal will be sent.
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C1_Init. Also, START signal needs to be issued in order to use this function. See I2C1_Start. Supported by the dsPIC33 and PIC24 MCUs only.

Example	`dim take as byte`
	`...`
	`' Read data and send the not_acknowledge signal`
	`take = I2C1_Read(1)`

I2C1_Write

Prototype	`sub function I2c1_Write(dim data as byte) as word`
Returns	- `0` if there were no errors. - `1` if write collision was detected on the I²C bus.
Description	Sends data byte via the I²C bus. Parameters : - `data`: data to be sent
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C1_Init. Also, START signal needs to be issued in order to use this function. See I2C1_Start. Supported by the dsPIC33 and PIC24 MCUs only.

Example	`dim data as byte`
	`  error as word`
	`...`
	`error = I2C1_Write(data)`
	`error = I2C1_Write(0xA3)`

I2C1_Stop

Prototype	`sub procedure I2C1_Stop()`
Description	Issues STOP signal.
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C1_Init. Supported by the dsPIC33 and PIC24 MCUs only.
Example	`' Issue STOP signal` `I2C1_Stop()`

I2C2_Init

Prototype	`sub procedure I2c2_Init(dim scl as longint)`
Description	Initializes the I²C2 module. This function enables the I2C2 module by setting the I2CEN bit. The rest of the bits in I2C2CON register remains unchanged. Default initialization (after reset) of the I2C2 module is: - continue operation in IDLE mode - IPMI mode disabled - 7-bit slave address - slew rate control enabled - general call address disabled - software or receive clock stretching disabled Parameters : - `scl`: requested serial clock rate. Refer to the MCU's datasheet for correct values of the scl in respect with *Fosc*.
Requires	MCU with the I²C2 module. Supported by the dsPIC33 and PIC24 MCUs only.
Example	`' Initialize the I2C2 module with clock_rate=100000` `I2C2_Init(100000)`

I2C2_Start

Prototype	`sub procedure I2c2_Start()`
Description	Determines if the I²C bus is free and issues START signal.
Requires	MCU with the I²C2 module. The I²C2 module must be initialized before using this function. See I2C2_Init. Supported by the dsPIC33 and PIC24 MCUs only.
Example	`' Issue START signal` `I2C2_Start()`

I2C2_Restart

Prototype	`sub procedure I2C2_Restart()`
Description	Issues repeated START signal.
Requires	MCU with the I²C2 module. The I²C2 module must be initialized before using this function. See I2C2_Init. Supported by the dsPIC33 and PIC24 MCUs only.
Example	`' Issue RESTART signal` `I2C2_Restart()`

I2C2_Wait_For_Idle

Prototype	`sub procedure I2C2_Wait_For_Idle()`
Description	Waits for the I²C bus to become free. This is a blocking function.
Requires	MCU with the I²C2 module. The I²C2 module must be initialized before using this function. See I2C2_Init. Supported by the dsPIC33 and PIC24 MCUs only.

Example	`dim data as byte`
	`...`
	`I2C2_Wait_For_Idle()`
	`I2C2_Write(data)`
	`...`

I2C2_Read

Prototype	`sub function I2c2_Read(dim ack as word) as byte`
Returns	Received data.
Description	Reads a byte from the I²C bus. Parameters : - `ack`: acknowledge signal parameter. If the `ack==0` acknowledge signal will be sent after reading, otherwise the *not acknowledge* signal will be sent.
Requires	MCU with the I²C2 module. The I²C2 module must be initialized before using this function. See I2C2_Init. Also, START signal needs to be issued in order to use this function. See I2C2_Start. Supported by the dsPIC33 and PIC24 MCUs only.

Example	`dim take as byte`
	`...`
	`' Read data and send the not_acknowledge signal`
	`take = I2C2_Read(1)`

I2C2_Write

Prototype	`sub function I2c2_Write(dim data as byte) as word`
Returns	- `0` if there were no errors. - `1` if write collision was detected on the I²C bus.
Description	Sends data byte via the I²C bus. Parameters : - `data`: data to be sent
Requires	MCU with the I²C2 module. The I²C2 module must be initialized before using this function. See I2C2_Init. Also, START signal needs to be issued in order to use this function. See I2C2_Start. Supported by the dsPIC33 and PIC24 MCUs only.

Example	`dim data as byte`
	`  error as word`
	`...`
	`error = I2C2_Write(data)`
	`error = I2C2_Write(0xA3)`

I2C2_Stop

Prototype	`sub procedure I2C2_Stop()`
Description	Issues STOP signal.
Requires	MCU with the I²C2 module. The I²C2 module must be initialized before using this function. See I2C2_Init. Supported by the dsPIC33 and PIC24 MCUs only.
Example	`' Issue STOP signal` `I2C2_Stop()`

I2C_Init

Prototype	`sub procedure I2c_Init(dim scl as longint)`
Description	Configures and initializes the I²C1 module with default settings. It is kept to maintain backwards compatibility with the previous versions of mikroBasic for dsPIC30/33 and PIC24. This function enables the I2C module by setting the I2CEN bit. The rest of the bits in I2C control register remains unchanged. Default initialization (after reset) of the I2C module is: - continue operation in IDLE mode - IPMI mode disabled - 7-bit slave address - slew rate control enabled - general call address disabled - software or receive clock stretching disabled Parameters : - `scl`: requested serial clock rate. Refer to the MCU's datasheet for correct values of scl in respect with *Fosc*.
Requires	MCU with the I²C1 module.
Example	`' Initialize the I2C1 module with clock_rate=100000` `I2C_Init(100000)`

I2C_Start

Prototype	`sub procedure I2C_Start()`
Description	Determines if the I²C1 bus is free and issues START signal.
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C_Init routine.
Example	`' Issue START signal` `I2C_Start()`

I2C_Restart

Prototype	`sub procedure I2C_Restart()`
Description	Issues repeated START signal.
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C_Init routine.
Example	`' Issue RESTART signal` `I2C_Restart()`

I2C_Wait_For_Idle

Prototype	`sub procedure I2C_Wait_For_Idle()`
Description	Waits for the I²C1 bus to become free. This is a blocking function.
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C_Init routine.

Example	`dim data as byte`
	`...`
	`I2C_Wait_For_Idle()`
	`I2C_Write(data)`
	`...`

I2C_Read

Prototype	`sub function I2c_Read(dim ack as word) as byte`
Returns	Received data.
Description	Reads a byte from the I²C1 bus. Parameters : - `ack`: acknowledge signal parameter. If the ack=0 acknowledge signal will be sent after reading, otherwise the not acknowledge signal will be sent.
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C_Init routine. Also, START signal needs to be issued in order to use this function. See the I2C_Start.

Example	`dim take as byte`
	`...`
	`' Read data and send the not_acknowledge signal`
	`take = I2C_Read(1)`

I2C_Write

Prototype	`sub function I2c_Write(dim data as byte) as word`
Returns	- `0` if there were no errors. - `1` if write collision was detected on the I²C bus.
Description	Sends data byte via the I²C1 bus. Parameters : - `data`: data to be sent
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C_Init routine. Also, START signal needs to be issued in order to use this function. See I2C_Start.

Example	`dim data as byte`
	`  error as word`
	`...`
	`error = I2C_Write(data)`
	`error = I2C_Write(0xA3)`

I2C_Stop

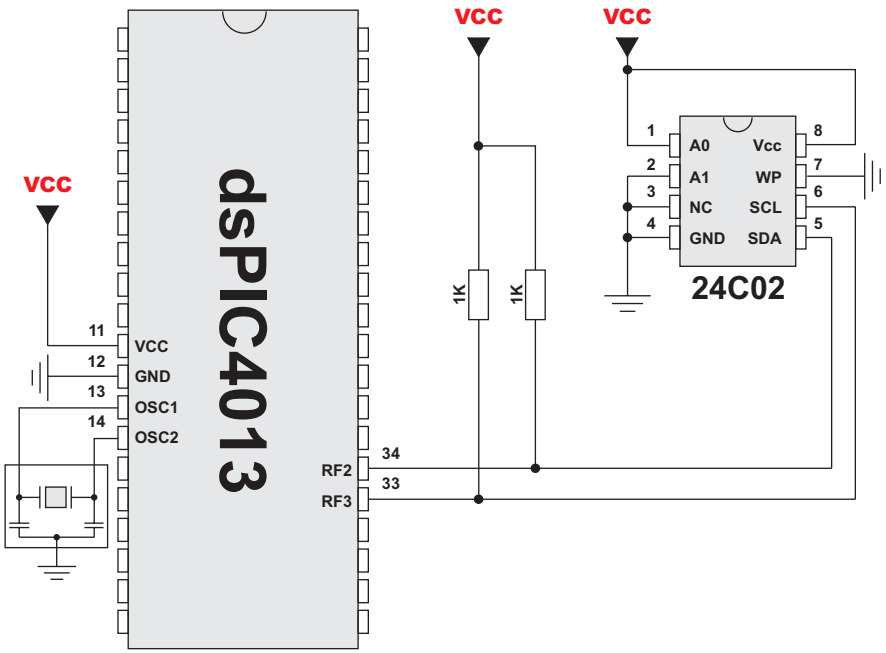
Prototype	`sub procedure I2C_Stop()`
Description	Issues STOP signal.
Requires	MCU with the I²C1 module. The I²C1 module must be initialized before using this function. See I2C_Init routine.
Example	`' Issue STOP signal` `I2C_Stop()`

Library Example

This code demonstrates the use of the I²C library. dsPIC30/33 and PIC24 MCU is connected (SCL, SDA pins) to 24c02 EEPROM. Program sends data to EEPROM (data is written at address 2). Then, we read data via I2C from EEPROM and send its value to PORTB, to check if the cycle was successful (see the figure below how to interface 24c02 to the dsPIC30/33 and PIC24).

```
program I2CEEPROM
dim dAddr as word

main:
  ADPCFG = 0xFFFF
  PORTB  = 0
  TRISB  = 0

  dAddr  = 0x02

  I2c_Init(0x000186A0)
  I2c_Start()                    ' issue I2C start signal
  I2c_Write(0xA2)                ' send byte via I2C  (command to 24c02)
  I2c_Write(dAddr)               ' send byte (address of EEPROM location)
  I2c_Write(0xF5)                ' send data (data to be written)
  I2c_Stop()

  Delay_ms(100)

  I2c_Start()                    ' issue I2C start signal
  I2c_Write(0xA2)                ' send byte via I2C  (device address + W)
  I2c_Write(dAddr)               ' send byte (data address)
  I2c_Restart()                  ' issue I2C signal repeated start
  I2c_Write(0xA3)                ' send byte (device address + R)
  PORTB = I2c_Read(1)            ' Read the data (NOT acknowledge)
  I2c_Stop()
end.
```

Hardware Connection



Keypad Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for working with 4x4 keypad. The library routines can also be used with 4x1, 4x2, or 4x3 keypad. For explanation of connections see schematic at the bottom of this page.

Library Routines

```
Keypad_Init
Keypad_Key_Press
Keypad_Key_Click
```

Keypad_Init

Prototype	`sub procedure Keypad_Init(dim byref port as word)`
Description	Initializes given port for working with keypad. Parameters : - `port`: keypad port. **Note**: The Keypad library uses lower byte (bits <7..0>) of `key_port`.
Example	`' Initialize PORTB<7:0> pins for communication with keypad` `Keypad_Init(PORTB)`

Keypad_Key_Press

Prototype	`sub function Keypad_Key_Press() as word`
Returns	The code of a pressed key (1..16), 0 if no key is pressed.
Description	Checks if any key is pressed and returns the key code.
Requires	Port needs to be initialized for working with the Keypad library, see Keypad_Init.

Example	`dim kp as word`
	`...`
	`kp = Keypad_Key_Press()`

Keypad_Key_Click

Prototype	`sub function Keypad_Key_Click() as word`
Returns	The code of a clicked key (1..16), 0 if no key is clicked.
Description	Call to `Keypad_Key_Click` is a blocking call: the function waits until some key is pressed *and* released. When released, the function returns 1 to 16, depending on the key. If more than one key is pressed simultaneously the function will wait until all pressed keys are released. After that the function will return the code of the first pressed key.
Requires	Port needs to be initialized for working with the Keypad library, see `Keypad_Init`.
Example	`kp = Keypad_Key_Click()`

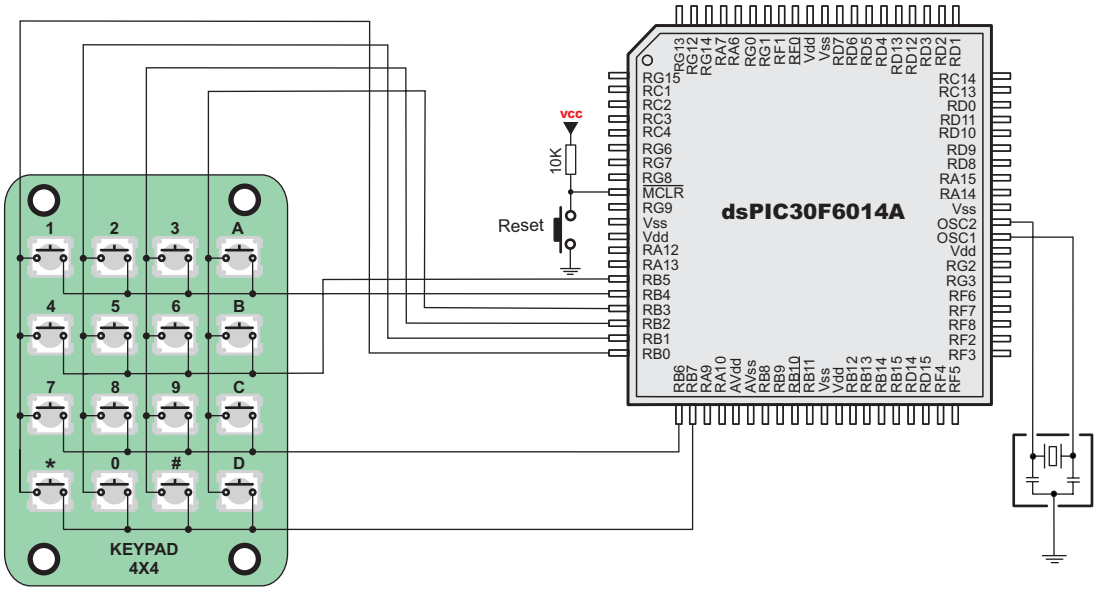
Library Example

The following code can be used for testing the keypad.

```
program Keypad_4x4_test

dim kp, oldkp  as byte
     cnt as word
     txt as string[ 6]

main:
  cnt = 0
  Keypad_Init(PORTF)

  ADPCFG = $FFFF

  Glcd_Init_dsPICPro3()
  Glcd_Fill(0x00)
  Glcd_Set_Font(@FontSystem5x8, 5, 8, 32)
  Glcd_Write_Text("Keypad 4x4 example", 10, 0, 1)
  Glcd_Write_Text("Key pressed :", 10, 3, 1)
  Glcd_Write_Text("Times :"       , 10, 6, 1)

  oldkp = 1

  while TRUE
    nop
    kp = 0
    ' Wait for key to be pressed
    while kp = 0
      kp = Keypad_Key_Click()
    wend

' continues ...
```

```
' .. continued

    if oldkp = kp then
      Inc(cnt)
    else
      cnt = 1
    end if

    oldkp = kp

    ' Prepare value for output
    select case kp
      case  1 kp = 49  ' 1
      case  2 kp = 50  ' 2
      case  3 kp = 51  ' 3
      case  4 kp = 65  ' A
      case  5 kp = 52  ' 4
      case  6 kp = 53  ' 5
      case  7 kp = 54  ' 6
      case  8 kp = 66  ' B
      case  9 kp = 55  ' 7
      case 10 kp = 56  ' 8
      case 11 kp = 57  ' 9
      case 12 kp = 67  ' C
      case 13 kp = 42  ' *
      case 14 kp = 48  ' 0
      case 15 kp = 35  ' #
      case 16 kp = 68  ' D
    end select

    Glcd_Write_Char(kp, 30, 4, 1)

    if cnt = 65535 then
      cnt = 0
    end if

    ' Print on LCD
    WordToStr(cnt, txt)
    Glcd_Write_Text(txt, 10, 7, 1)

  wend ' while

end.
```

Hardware Connection



LCD Custom Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for communication with LCDs (with HD44780 compliant controllers) through the 4-bit interface. An example of LCD connections is given on the schematic at the bottom of this page.

For creating a custom set of LCD characters use the LCD Custom Character Tool.

Note: mikroElektronika's development system based initialization routines can be found in the setup library files located in the Uses folder.

Note: Only the Lcd_Custom_Config routine uses the RW pin (RW pin is configured as output and set to zero). If the user needs this pin for other purposes, it can be reconfigured after the Lcd_Custom_Config call.

Library Routines

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

Lcd_Init

Prototype	`sub procedure Lcd_Init(dim byref data_port as word, dim db3, db2, db1, db0 as word, dim byref ctrl_port as word, dim rs, ctrl_rw, enable as word)`
Description	Initializes LCD with custom pin settings. Parameters : - `data_port`: data port - `db3`: data bit 3 - `db2`: data bit 2 - `db1`: data bit 1 - `db0`: data bit 0 - `ctrl_port`: control port - `rs`: register select (data/instruction) signal pin - `ctrl_rw`: read/write signal pin - `enable`: enable signal pin

Example	`' Init for EasydsPIC3 development system`
	`Lcd_Init(PORTB, 3,2,1,0, PORTD, 0,2,1)`
	
	`' or just call the appropriate system based`
	`' initialization routine`
	`Lcd_Init_EasyDsPIC3()`

Lcd_Out

Prototype	`sub procedure Lcd_Out(dim row, column as byte, dim byref text as string[ 20] )`
Description	Prints text on LCD starting from specified position. Both string variables and literals can be passed as a text. Parameters : - `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Requires	The LCD module needs to be initialized. See the Lcd_Init routine.
Example	`' Write text "Hello!" on LCD starting from row 1, column 3:` `Lcd_Out(1, 3, "Hello!")`

Lcd_Out_Cp

Prototype	`sub procedure Lcd_Out_Cp(dim byref text as string[20])`
Description	Prints text on LCD at current cursor position. Both string variables and literals can be passed as a text. Parameters : - `text`: text to be written
Requires	The LCD module needs to be initialized. See the Lcd_Init routine.
Example	`' Write text "Here!" at current cursor position:` `Lcd_Out_Cp("Here!")`

Lcd_Chr

Prototype	`sub procedure Lcd_Chr(dim row, column, out_char as byte)`
Description	Prints character on LCD at specified position. Both variables and literals can be passed as a character. Parameters : - `row`: writing position row number - `column`: writing position column number - `out_char`: character to be written
Requires	The LCD module needs to be initialized. See the Lcd_Init routine.
Example	`' Write character "i" at row 2, column 3:` `Lcd_Chr(2, 3, "i")`

Lcd_Chr_Cp

Prototype	`sub procedure Lcd_Chr_Cp(dim out_char as byte)`
Description	Prints character on LCD at current cursor position. Both variables and literals can be passed as a character. Parameters : - `out_char`: character to be written
Requires	The LCD module needs to be initialized. See the Lcd_Init routine.
Example	`' Write character "e" at current cursor position:` `Lcd_Chr_Cp("e")`

Lcd_Cmd

Prototype	`sub procedure Lcd_Cmd(dim out_char as byte)`
Description	Sends command to LCD. Parameters : - `out_char`: command to be sent **Note**: Predefined constants can be passed to the function, see Available LCD Commands.
Requires	The LCD module needs to be initialized. See the Lcd_Init routine.
Example	`' Clear LCD display:` `Lcd_Cmd(LCD_CLEAR)`

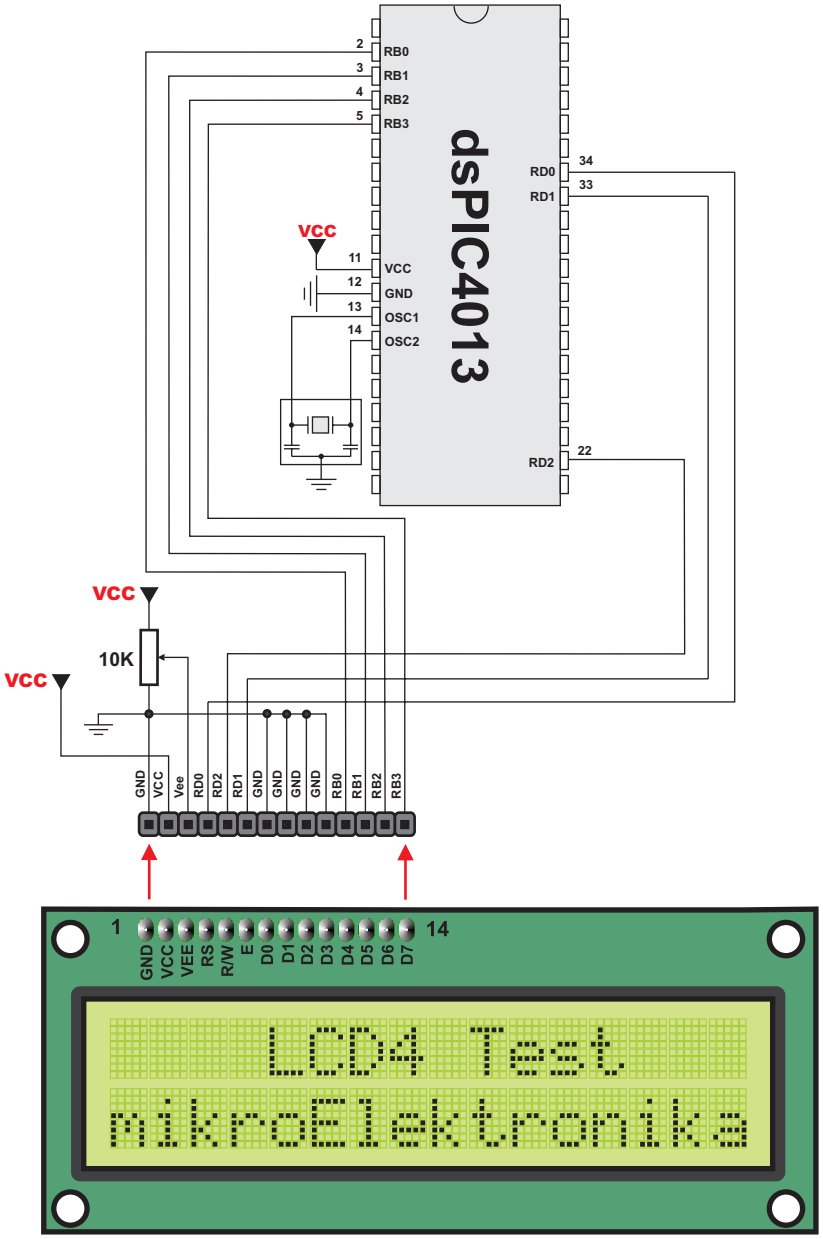
LCD Commands

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

```
program LCD4bitTest

dim txt as string[ 15]

main:
  ADPCFG = $FFFF
  Lcd_Init_EasydsPIC4()

  Lcd_Cmd(LCD_CURSOR_OFF)
  Lcd_Cmd(LCD_FIRST_ROW)

  LCD_Chr_Cp("m")  ' m
  LCD_Chr_Cp(105)  ' i
  LCD_Chr_Cp(107)  ' k
  LCD_Chr_Cp("r")  ' r
  LCD_Chr_Cp(111)  ' o

  Lcd_Cmd(LCD_SECOND_ROW)
  Lcd_Out(2, 6, "Elektronika")

  Delay_ms(1000)

  txt = "Elektronika"
  Lcd_Out(1, 6, txt)

end.
```

Hardware Connection



LCD 2X16

LCD8 (8-bit interface) Custom Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for communication with LCDs (with HD44780 compliant controllers) through the 8-bit interface. An example of LCD connections is given on the schematic at the bottom of this page.

For creating a custom set of LCD characters use the LCD Custom Character Tool.

Note: mikroElektronika's development system based initialization routines can be found in the setup library files located in the Uses folder.

Note: Only the `Lcd8_Init` and `Lcd8_Config` routines use the RW pin (RW pin is configured as output and set to zero). If the user needs this pin for other purposes, it can be reconfigured after the `Lcd8_Init` or `Lcd8_Config` call.

Library Routines

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

Lcd8_Init

Prototype	`sub procedure Lcd8_Init(dim byref data_port as word, dim db7, db6, db5, db4, db3, db2, db1, db0 as word, dim byref ctrl_port as word, dim rs, ctrl_rw, enable as word)`
Description	Initializes LCD with custom pin settings. Parameters : - `data_port`: data port - `db7`: data bit 7 - `db6`: data bit 6 - `db5`: data bit 5 - `db4`: data bit 4 - `db3`: data bit 3 - `db2`: data bit 2 - `db1`: data bit 1 - `db0`: data bit 0 - `ctrl_port`: control port - `rs`: register select (data/instruction) signal pin - `ctrl_rw`: read/write signal pin - `enable`: enable signal pin

Example	`' Init for EasydsPIC2 development system`
	`Lcd8_Init(PORTB, 7, 6, 5, 4, 3, 2, 1, 0,PORTD, 0, 1, 2)`
	
	`' or just call the appropriate system based`
	`' initialization routine`
	`Lcd8_Init_EasyDsPIC2()`

Lcd8_Config

Prototype	`sub procedure Lcd8_Config(dim byref data_portHi as word, dim db7, db6, db5, db4 as word, dim byref data_portLo as word, dim db3, db2, db1, db0 as word, dim byref ctrl_port as word, dim rs, ctrl_rw, enable as word)`
Description	Initializes LCD with custom pin settings. LCD data lines can be configured on two MCU ports. Parameters : - `data_portHi`: MCU's port connected to LCD's D7..D4 data lines - `db7`: data bit 7 - `db6`: data bit 6 - `db5`: data bit 5 - `db4`: data bit 4 - `data_portLo`: MCU's port connected to LCD's D3..D0 data lines - `db3`: data bit 3 - `db2`: data bit 2 - `db1`: data bit 1 - `db0`: data bit 0 - `ctrl_port`: control port - `rs`: register select (data/instruction) signal pin - `ctrl_rw`: read/write signal pin - `enable`: enable signal pin

Example	`' Init for EasydsPIC3 development system`
	`Lcd8_Config(PORTD, 3, 2, 1, 0,PORTB, 3, 2, 1, 0,PORTF, 0, 1, 4)`
	
	`' or just call the appropriate system based`
	`' initialization routine`
	`Lcd8_Init_EasyDsPIC3()`

Lcd8_Out

Prototype	`sub procedure Lcd8_Out(dim row, column as word, dim byref text as string[20])`
Description	Prints text on LCD starting from the specified position. Both string variables and literals can be passed as a text. Parameters : - `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Requires	The LCD module needs to be initialized. See the Lcd8_Init and Lcd8_Config routines.
Example	`' Write text "Hello!" on LCD starting from row 1, column 3:` `Lcd8_Out(1, 3, "Hello!")`

Lcd8_Out_Cp

Prototype	`sub procedure Lcd8_Out_Cp(dim byref text as string[20])`
Description	Prints text on LCD at current cursor position. Both string variables and literals can be passed as a text. Parameters : - `text`: text to be written
Requires	The LCD module needs to be initialized. See the Lcd8_Init and Lcd8_Config routines.
Example	`' Write text "Here!" at current cursor position:` `Lcd8_Out_Cp("Here!")`

Lcd8_Chr

Prototype	`sub procedure Lcd8_Chr(dim row, column, out_char as byte)`
Description	Prints character on LCD at the specified position. Both variables and literals can be passed as a character. Parameters : - `row`: writing position row number - `column`: writing position column number - `out_char`: character to be written
Requires	The LCD module needs to be initialized. See the Lcd8_Init and Lcd8_Config routines.
Example	`' Write character "i" at row 2, column 3:` `Lcd8_Out(2, 3, "i")`

Lcd8_Chr_Cp

Prototype	`sub procedure Lcd8_Chr_Cp(dim out_char as byte)`
Description	Prints character on LCD at current cursor position. Both variables and literals can be passed as a character. Parameters : - `out_char`: character to be written
Requires	The LCD module needs to be initialized. See the Lcd8_Init and Lcd8_Config routines.
Example	`' Write character "e" at current cursor position:` `Lcd8_Chr_Cp("e")`

Lcd8_Cmd

Prototype	`sub procedure Lcd8_Cmd(dim out_char as word)`
Description	Sends command to LCD. Parameters : - `out_char`: command to be sent Note: Predefined constants can be passed to the function, see Available LCD Commands.
Requires	The LCD module needs to be initialized. See the Lcd8_Init and Lcd8_Config routines.
Example	`' Clear LCD display:` `Lcd8_Cmd(LCD_CLEAR)`

Library Example

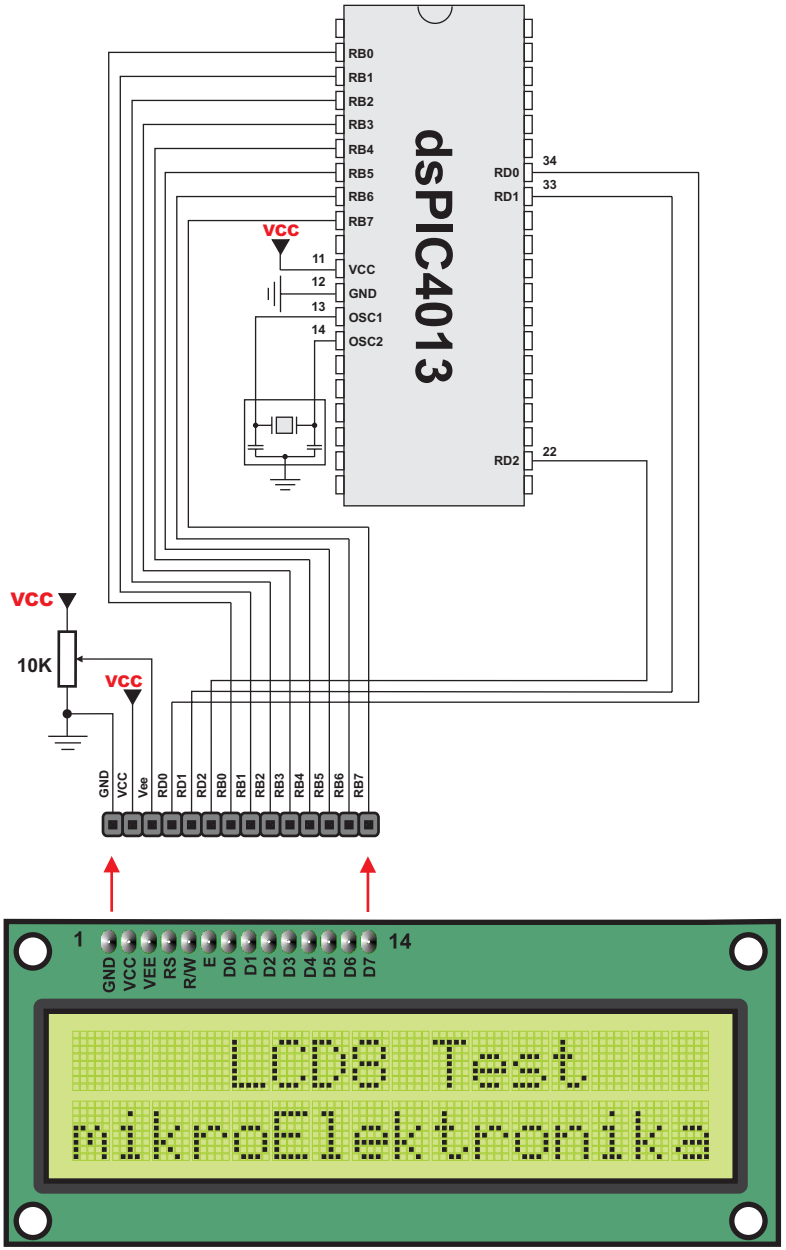
Here is an example of using the Lcd8 Custom Library.

```
program Lcd8Bit_Test

dim txt as string[ 15]

main:
  ADPCFG = $FFFF                          ' PORTB to be digital
  Lcd8_Init(PORTB, 7, 6, 5, 4, 3, 2, 1, 0, PORTD, 0, 1, 2)
  Lcd8_Cmd(LCD_CURSOR_OFF)
  Lcd8_Cmd(LCD_FIRST_ROW)

  LCD8_Chr_Cp("m") ' m
  LCD8_Chr_Cp(105) ' i
  LCD8_Chr_Cp(107) ' k
  LCD8_Chr_Cp("r") ' r
  LCD8_Chr_Cp(111) ' o

  Lcd8_Cmd(LCD_SECOND_ROW)
  Lcd8_Out(2, 6, "Elektronika")
  Delay_ms(1000)
  txt = "Elektronika"
  Lcd8_Out(1, 6, txt)

end.
```

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

Hardware Connection



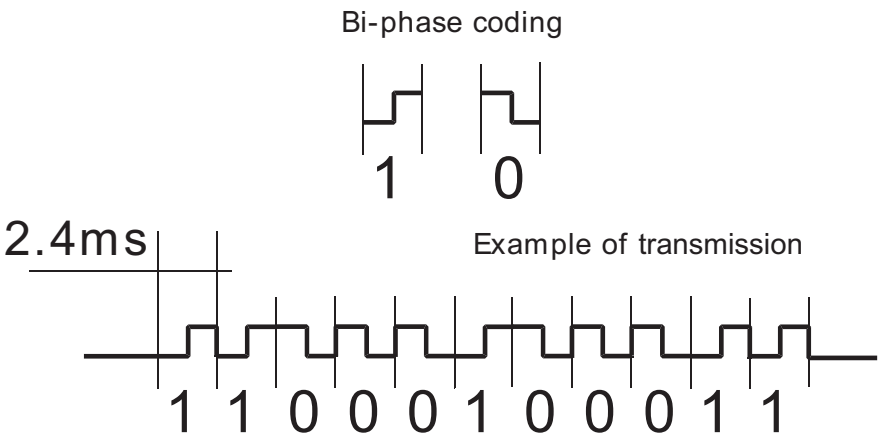
LCD 2X16

Manchester Code Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for handling Manchester coded signals. The Manchester code is a code in which data and clock signals are combined to form a single self-synchronizing data stream; each encoded bit contains a transition at the midpoint of a bit period, the direction of transition determines whether the bit is 0 or 1; the second half is the true bit value and the first half is the complement of the true bit value (as shown in the figure below).

Manchester RF_Send_Byte format

St1	St2	Ctr	B7	B6	B5	B4	B3	B2	B1	B0



Notes: The Manchester receive routines are blocking calls (`Man_Receive_Config`, `Man_Receive_Init` and `Man_Synchro`). This means that MCU will wait until the task has been performed (e.g. byte is received, synchronization achieved, etc).

Library Routines

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

The following routines are for the internal use by the compiler only:

```
Manchester_0
Manchester_1
Manchester_Out
```

Man_Receive_Config

Prototype	`sub function Man_Receive_Config(dim byref port as word, dim rxpin as word) as word`
Returns	- `0` - if initialization and synchronization were successful. - `1` - upon unsuccessful synchronization.
Description	The function configures Receiver pin and performs synchronization procedure in order to retrieve baud rate out of incoming signal. Parameters : - `port`: Receiver port - `rxpin`: Receiver pin **Note**: In case of multiple persistent errors on reception, the user should call this routine once again or Man_Synchro routine to enable synchronization.
Example	`' Configure Receiver on pin RD3:` `Man_Receive_Config(PORTD, 3)`

Man_Receive_Init

Prototype	`sub function Man_Receive_Init(dim byref port as word) as word`
Returns	- `0` - if initialization and synchronization were successful. - `1` - upon unsuccessful synchronization
Description	The function configures Receiver pin as pin3 of the MCU's port. After that, the function performs synchronization procedure in order to retrieve baud rate out of incoming signal. Parameters : - `port`: Receiver port **Note**: In case of multiple persistent errors on reception, the user should call this routine once again or Man_Synchro routine to enable synchronization.
Example	`' Configure Receiver on pin RD3:` `Man_Receive_Init(PORTD)`

Man_Receive

Prototype	`sub function Man_Receive(dim byref error as word) as byte`
Returns	A byte read from the incoming signal.
Description	The function extracts one byte from incoming signal. Parameters : - `error`: error flag. If signal format does not match the expected, the error flag will be set to non-zero.
Requires	To use this function, the user must prepare the MCU for receiving. See Man_Receive_Config or Man_Receive_Init.

Example	`dim data, error as word`
	`...`
	`error = 0`
	`data = 0`
	`data = Man_Receive(error)`
	`if (error <> 0) then`
	`  ' error handling`
	`end if`

Man_Send_Config

Prototype	`sub procedure Man_Send_Config(dim byref port as word, dim txpin as word)`
Description	The function configures Transmitter pin. Parameters : - `port`: Transmitter port - `txpin`: Transmitter pin
Example	`' Configure Transmitter on pin RD0:` `Man_Send_Config(PORTD, 0)`

Man_Send_Init

Prototype	`sub procedure Man_Send_Init(dim byref port as word)`
Description	The function configures Transmitter pin as pin0 of the MCU's port. Parameters : - `port`: Transmitter port
Example	`' Configure Transmitter on pin RD0:` `Man_Send_Init(PORTD)`

Man_Send

Prototype	`sub procedure Man_Send(dim data as byte)`
Description	Sends one byte. Parameters : - `data`: data to be sent **Note**: Baud rate used is 500 bps.
Requires	To use this function, the user must prepare the MCU for sending. See Man_Send_Config or Man_Send_Init.

Example	`dim msg as byte`
	`...`
	`Man_Send(msg)`

Man_Synchro

Prototype	`sub function Man_Synchro() as word`
Returns	- `0` - if synchronization was not successful. - Half of the manchester bit length, given in multiples of 10us upon successful synchronization.
Description	Measures half of the manchester bit length with 10us resolution.
Requires	To use this function, you must first prepare the MCU for receiving. See Man_Receive_Config or Man_Receive_Init.

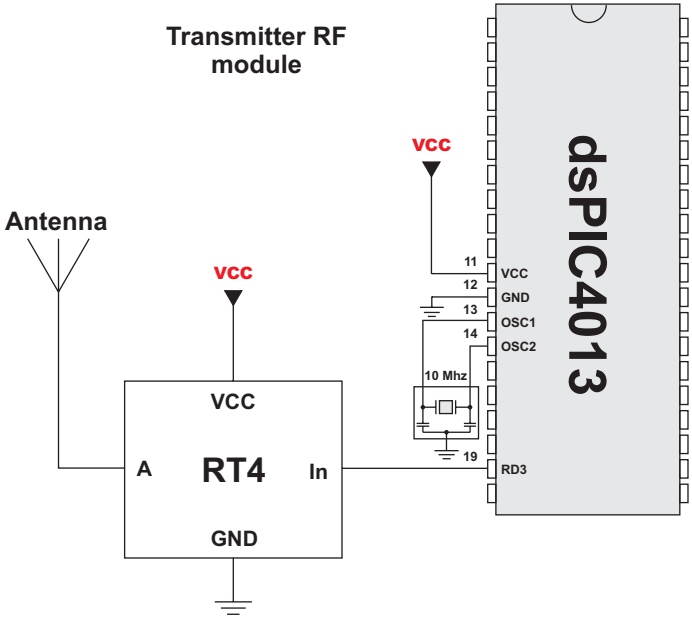
Example	`dim man__half_bit_len as word`
	`...`
	`man__half_bit_len = Man_Synchro()`

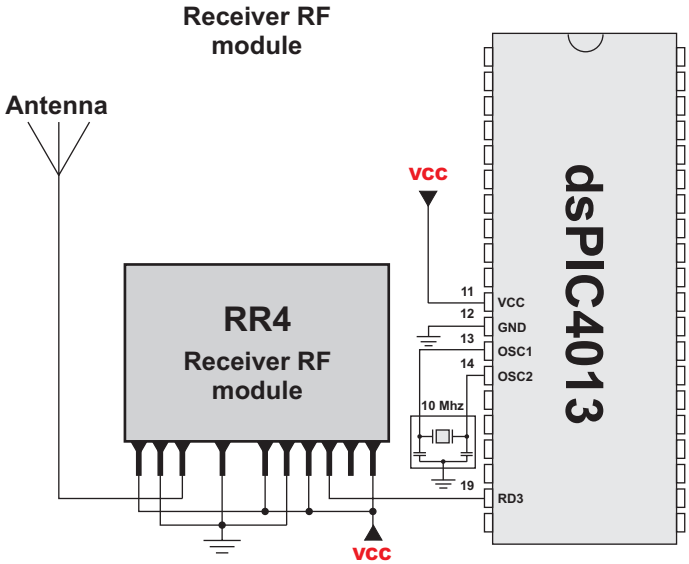
Library Example

The following code receives message in Manchester code:

```
program Manchester_Rcv

dim
     _error,
     ErrorCount,
     tmp as word

main:
  ADPCFG = 0xFFFF
  _error = 0
  ErrorCount = 0

  Lcd_Init_EasydsPIC4()                    ' Initialize LCD on EasydsPIC4
  Lcd_Out(1,1,"mikroManchester")
  Man_Receive_Config(PORTC, 13)            ' Configure and synchronize receiver
  while true
    Lcd_Cmd(LCD_FIRST_ROW)
    while true                               ' Wait for the start marker
      tmp = Man_Receive(_error)
      if (tmp = 0x0B) then
        break                                 ' We got the starting sequence
      end if
      if(_error <> 0) then                   ' Exit so we do not loop forever
        break
      end if
    wend

    while(tmp <> 0x0E)
      tmp = Man_Receive(_error)            ' Attempt byte receive
      if (_error <> 0) then
        Lcd_Chr_CP("?")
        ErrorCount = ErrorCount + 1
        if(ErrorCount > 20) then
          'Man_Receive_Init(PORTD)
          ' alternative:
          tmp = Man_Synchro()
          ErrorCount = 0
        end if
      else
        if (tmp <> 0x0E) then              ' Don't write the end marker on LCD
          Lcd_Chr_CP(tmp)
        end if
      end if
      Delay_ms(25)
    wend
  wend
```

Hardware Connection





Multi Media Card Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for handling data on MMC and SD memory cards using SPI interface.

Note: Routines for file handling can be used only with FAT16 file system.
Note: Library functions create and read files from the root directory only.
Note: Library functions populate both FAT1 and FAT2 tables when writing to files, but the file data is being read from the FAT1 table only; i.e. there is no recovery if the FAT1 table gets corrupted.

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

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

Note: Library uses SPI module for communication. The user must initialize the appropriate SPI module before using the MMC Library. For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `Spi_Set_Active()` function. See the Spi Library functions.

The SPI module has to be initialized through the `Spi_Init_Advanced` routine with the following parameters:

- SPI Master
- 8bit mode
- secondary prescaler 1
- primary prescaler 64
- Slave Select disabled
- data sampled in the middle of data output time
- clock idle high
- Serial output data changes on transition from active clock state to idle clock state

Note: Once the MMC/SD card is initialized, the user can reinitialize SPI at higher speed. See the Mmc_Init and Mmc_Fat_Init routines.

Library Routines

```
Mmc_Init                    Mmc_Fat_Read
Mmc_Read_Sector             Mmc_Fat_Rewrite
Mmc_Write_Sector            Mmc_Fat_Append
Mmc_Read_Cid                Mmc_Fat_Delete
Mmc_Read_Csd                Mmc_Fat_Write
Mmc_Fat_Init                Mmc_Fat_Set_File_Date
Mmc_Fat_QuickFormat         Mmc_Fat_Get_File_Date
Mmc_Fat_Assign              Mmc_Fat_Get_File_Size
Mmc_Fat_Reset               Mmc_Fat_Get_Swap_File
```

Mmc_Init

Prototype	`sub function Mmc_Init(dim byref port as word, dim cspin as word) as word`
Returns	- `0` - if MMC/SD card was detected and successfully initialized - `1` - otherwise
Description	Initializes MMC through hardware SPI interface. Parameters: - `port`: chip select signal port address. - `cspin`: chip select pin.
Requires	The appropriate hardware SPI module must be previously initialized. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.

Example	`dim i as word`
	`...`
	`' Initialize the SPI module`
	`Spi1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_64, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE)`
	`' MMC initialization`
	
	`i = Mmc_Init(PORTF, 0)`

Mmc_Read_Sector

Prototype	`sub function Mmc_Read_Sector(dim sector as longint, dim byref data as byte[ 512] ) as word`
Returns	- 0 - if reading was successful - 1 - if an error occurred
Description	The function reads one sector (512 bytes) from MMC card. Parameters: - sector: MMC/SD card sector to be read. - data: buffer of minimum 512 bytes in length for data storage.
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	`' read sector 510 of the MMC/SD card`
	`dim error as word`
	`  sectorNo as longint`
	`  dataBuffer as byte[ 512]`
	`...`
	`sectorNo = 510`
	`error = Mmc_Read_Sector(sectorNo, dataBuffer)`

Mmc_Write_Sector

Prototype	`sub function Mmc_Write_Sector(dim sector as longint, dim byref data as byte[ 512] ) as word`
Returns	- 0 - if writing was successful - 1 - if there was an error in sending write command - 2 - if there was an error in writing (data rejected)
Description	The function writes 512 bytes of data to one MMC card sector. Parameters: - sector: MMC/SD card sector to be written to. - data: data to be written (buffer of minimum 512 bytes in length).
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	`' write to sector 510 of the MMC/SD card`
	`dim error as word`
	`  sectorNo as longint`
	`  dataBuffer as byte[ 512]`
	`...`
	`sectorNo = 510`
	`error = Mmc_Write_Sector(sectorNo, dataBuffer)`

Mmc_Read_Cid

Prototype	`sub function Mmc_Read_Cid(dim byref data_cid as byte[ 2] ) as word`
Returns	- `0` - if CID register was read successfully - `1` - if there was an error while reading
Description	The function reads 16-byte CID register. Parameters: - `data_cid`: buffer of minimum 16 bytes in length for storing CID register content.
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	`dim error as word`
	`  dataBuffer as byte[ 512]`
	`...`
	`error = Mmc_Read_Cid(dataBuffer)`

Mmc_Read_Csd

Prototype	`sub function Mmc_Read_Csd(dim byref data_csd as byte[ 16] ) as word`
Returns	- `0` - if CSD register was read successfully - `1` - if there was an error while reading
Description	The function reads 16-byte CSD register. Parameters: - `data_csd`: buffer of minimum 16 bytes in length for storing CSD register content.
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	`dim error as word`
	`  dataBuffer as byte[ 512]`
	`...`
	`error = Mmc_Read_Csd(dataBuffer)`

Mmc_Fat_Init

Prototype	`sub function Mmc_Fat_Init(dim byref port as word, dim pin as word) as word`
Returns	- `0` - if MMC/SD card was detected and successfully initialized - `1` - if FAT16 boot sector was not found - `255` - if MMC/SD card was not detected
Description	Initializes MMC/SD card, reads MMC/SD FAT16 boot sector and extracts necessary data needed by the library. Parameters: - `port`: chip select signal port address. - `pin`: chip select pin. **Note**: MMC/SD card has to be formatted to FAT16 file system.
Requires	The appropriate hardware SPI module must be previously initialized.

Example	`' Initialize the SPI module`
	`Spi1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_64, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE)`
	`' Initialize MMC/SD card and MMC_FAT16 library globals`
	`if (Mmc_Fat_Init(PORTF, 0) = 0) then ' EasydsPIC4 board`
	`  ' reinitialize spi at higher speed`
	`  Spi1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_4, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE)`
	`  ...`
	`end if`

Mmc_Fat_QuickFormat

Prototype	`sub function Mmc_Fat_QuickFormat(dim byref port as word, dim pin as word, dim byref mmc_fat_label as string[11])) as word`
Returns	- `0` - if MMC/SD card was detected, successfully formated and initialized - `1` - if FAT16 format was unseccessful - `255` - if MMC/SD card was not detected
Description	Formats to FAT16 and initializes MMC/SD card. Parameters: - `port`: chip select signal port address. - `pin`: chip select pin. - `mmc_fat_label`: volume label (11 characters in length). If less than 11 characters are provided, the label will be padded with spaces. If an empty string is passed, the volume will not be labeled. **Note**: This routine can be used instead or in conjunction with the Mmc_Fat_Init routine. **Note**: If MMC/SD card already contains a valid boot sector, it will remain unchanged (except volume label field) and only FAT and ROOT tables will be erased. Also, the new volume label will be set.
Requires	The appropriate hardware SPI module must be previously initialized.

Example	`' Initialize the SPI module`
	`Spi1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_64, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE)`
	
	`' Format and initialize MMC/SD card and MMC_FAT16 library globals`
	`Mmc_Fat_QuickFormat(PORTF, 0, "mikroE")`
	
	`' Reinitialize the SPI module at higher speed`
	`' (change primary prescaler).`
	`Spi1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_4, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE)`

Mmc_Fat_Assign

Prototype	`sub function Mmc_Fat_Assign(dim byref filename as string[11], dim file_cre_attr as byte) as word`
Returns	- `1` - if file already exists or file does not exist but a new file is created. - `0` - if file does not exist and no new file is created.
Description	Assigns file for file operations (read, write, delete...). All subsequent file operations will be applied on an assigned file. Parameters: - `filename`: name of the file that should be assigned for file operations. File name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between file name and extension (i.e. "MIKROELETXT" -> MIKROELE.TXT). In this case last 3 characters of the string are considered to be file extension. - `file_cre_attr`: file creation and attributs flags. Each bit corresponds to the appropriate file attribut:

BIT	MASK	Description
0	0x01	Read Only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume Label
4	0x10	Subdirectory
5	0x20	Archive
6	0x40	Device (internal use only, never found on disk)
7	0x80	File creation flag. If the file does not exist and this flag is set, a new file with specified name will be created.

Note: Long File Names (LFN) are not supported.

Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init.
Example	`' create file with archive attribut if it does not already exist` `Mmc_Fat_Assign("MIKRO007.TXT",0xA0)`

Mmc_Fat_Reset

Prototype	`sub procedure Mmc_Fat_Reset(dim byref size as longint)`
Description	Opens currently assigned file for reading. Parameters: - `size`: buffer to store file size to. After file has been open for reading, its size is returned through this parameter.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`dim size as longint`
	`...`
	`Mmc_Fat_Reset(size)`

Mmc_Fat_Read

Prototype	`sub procedure Mmc_Fat_Read(dim byref bdata as byte)`
Description	Reads a byte from the currently assigned file opened for reading. Upon function execution, file pointers will be set to the next character in the file. Parameters: - `bdata`: buffer to store read byte to. Upon this function execution read byte is returned through this parameter.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign. The file must be opened for reading. See Mmc_Fat_Reset.

Example	`dim character as byte`
	`...`
	`Mmc_Fat_Read(character)`

Mmc_Fat_Rewrite

Prototype	`sub procedure Mmc_Fat_Rewrite()`
Description	Opens the currently assigned file for writing. If the file is not empty its content will be erased.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.
Example	`' open file for writing` `Mmc_Fat_Rewrite()`

Mmc_Fat_Append

Prototype	`sub procedure Mmc_Fat_Append()`
Description	Opens the currently assigned file for appending. Upon this function execution file pointers will be positioned after the last byte in the file, so any subsequent file writing operation will start from there.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.
Example	`' open file for appending` `Mmc_Fat_Append()`

Mmc_Fat_Delete

Prototype	`sub procedure Mmc_Fat_Delete()`
Description	Deletes currently assigned file from MMC/SD card.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.
Example	`' delete current file` `Mmc_Fat_Delete()`

Mmc_Fat_Write

Prototype	`sub procedure Mmc_Fat_Write(dim byref fdata as byte[ 12] , dim data_len as word)`
Description	Writes requested number of bytes to the currently assigned file opened for writing. Parameters: - `fdata`: data to be written. - `data_len`: number of bytes to be written.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign. The file must be opened for writing. See Mmc_Fat_Rewrite or Mmc_Fat_Append.

Example	`dim file_contents as byte[ 42]`
	`...`
	`Mmc_Fat_Write(file_contents, 42)`
	`' write data to the assigned file`

Mmc_Set_File_Date

Prototype	`sub procedure Mmc_Fat_Set_File_Date(dim year as word, dim month as byte, dim day as byte, dim hours as byte, dim mins as byte, dim seconds as byte)`
Description	Sets the date/time stamp. Any subsequent file writing operation will write this stamp to the currently assigned file's time/date attributs. Parameters: - `year`: year attribute. Valid values: 1980-2107 - `month`: month attribute. Valid values: 1-12 - `day`: day attribute. Valid values: 1-31 - `hours`: hours attribute. Valid values: 0-23 - `mins`: minutes attribute. Valid values: 0-59 - `seconds`: seconds attribute. Valid values: 0-59
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign. The file must be opened for writing. See Mmc_Fat_Rewrite or Mmc_Fat_Append.
Example	`Mmc_Fat_Set_File_Date(2005,9,30,17,41,0)`

Mmc_Fat_Get_File_Date

Prototype	`sub procedure Mmc_Fat_Get_File_Date(dim byref year as word, dim byref month as byte, dim byref day as byte, dim byref hours as byte, dim byref mins as byte)`
Description	Reads time/date attributes of the currently assigned file. Parameters: - `year`: buffer to store year attribute to. Upon function execution year attribute is returned through this parameter. - `month`: buffer to store month attribute to. Upon function execution month attribute is returned through this parameter. - `day`: buffer to store day attribute to. Upon function execution day attribute is returned through this parameter. - `hours`: buffer to store hours attribute to. Upon function execution hours attribute is returned through this parameter. - `mins`: buffer to store minutes attribute to. Upon function execution minutes attribute is returned through this parameter.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`dim year as word`
	`  month, day, hours, mins as byte`
	`...`
	`Mmc_Fat_Get_File_Date(year, month, day, hours, mins)`

Mmc_Fat_Get_File_Size

Prototype	`sub function Mmc_Fat_Get_File_Size() as longint`
Returns	Size of the currently assigned file in bytes.
Description	This function reads size of the currently assigned file in bytes.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`dim my_file_size as longint`
	`...`
	`my_file_size = Mmc_Fat_Get_File_Size()`

Mmc_Fat_Get_Swap_File

Prototype	`sub function Mmc_Fat_Get_Swap_File(dim sectors_cnt as longint, dim byref filename as string[11], dim file_attr as byte) as longint`
Returns	- Number of the start sector for the newly created swap file, if there was enough free space on the MMC/SD card to create file of required size. - `0` - otherwise.
Description	This function is used to create a swap file of predefined name and size on the MMC/SD media. If a file with specified name already exists on the media, search for consecutive sectors will ignore sectors occupied by this file. Therefore, it is recommended to erase such file if it already exists before calling this function. If it is not erased and there is still enough space for a new swap file, this function will delete it after allocating new memory space for a new swap file. The purpose of the swap file is to make reading and writing to MMC/SD media as fast as possible, by using the Mmc_Read_Sector() and Mmc_Write_Sector() functions directly, without potentially damaging the FAT system. The swap file can be considered as a "window" on the media where the user can freely write/read data. Its main purpose in the mikroBasic's library is to be used for fast data acquisition; when the time-critical acquisition has finished, the data can be re-written into a "normal" file, and formatted in the most suitable way. Parameters: - `sectors_cnt`: number of consecutive sectors that user wants the swap file to have. - `filename`: name of the file that should be assigned for file operations. File name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between file name and extension (i.e. "MIKROELETXT" -> MIKROELE.TXT). In this case the last 3 characters of the string are considered to be file extension. - `file_attr`: file creation and attributs flags. Each bit corresponds to the appropriate file attribut: *' continues on the next page ...*

```
' continued from the previous page ...
```

BIT	MASK	Description
0	0x01	Read Only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume Label
4	0x10	Subdirectory
5	0x20	Archive
6	0x40	Device (internal use only, never found on disk)
7	0x80	Not Used

Note: Long File Names (LFN) are not supported.

Requires

MMC/SD card and MMC library must be initialized for file operations.
See Mmc_Fat_Init.

Example

```
' Try to create a swap file with archive atribute, whose size
' will be at least 1000 sectors.
' If it succeeds, it sends No. of start sector over USART

dim size as longint
...
size = Mmc_Fat_Get_Swap_File(1000, "mikroE.txt", 0x20)
if (size <> 0) then
  Usart_Write(0xAA)
  Usart_Write(Lo(size))
  Usart_Write(Hi(size))
  Usart_Write(Higher(size))
  Usart_Write(Highest(size))
  Usart_Write(0xAA)
end if
```

Library Example

The following example demonstrates usage of the mmc and mmc_fat routines. Hardware configurations in this example are made for the EasydsPIC3 board and dsPIC30F4013.

```
program MMC_Fat16_Test

const
  SWAP_FILE_MSG as string[15] = "Swap file at: "

dim
  fat_txt as string[20]
  file_contents as string[50]

 filename as byte[14]  ' File names

 loopP, loop2 as word
 caracter as byte
 i, size as longint

 Buffer as byte[512]

sub procedure Glob_Init
 fat_txt = "FAT16 not found"
 file_contents = "XX MMC/SD FAT16 library by Anton Rieckert" + Chr(10) + Chr(0)
 filename = "MIKRO00xTXT" ' File names
end sub

'I-I-I--------- Writes string to USART
sub procedure I_Write_Str(dim byref ostr as byte[2])
dim  i as word
  i = 0
  while ostr[i] <> 0
    Uart1_Write_Char(ostr[i])
    Inc(i)
  wend
end sub '~

' continues ...
```

```
' continued ...

'M-M-M--------- Creates new file and writes some data to it
sub procedure M_Create_New_File
  filename[ 7]  = "A"
  Mmc_Fat_Assign(filename, 0xA0)    ' Will not find file and then create file
  Mmc_Fat_Rewrite                   ' To clear file and start with new data
  for loopP = 1 to 99               '  We want 5 files on the MMC card
      Uart1_Write_Char(".")
      file_contents[ 0]  = (loopP div 10) + 48
      file_contents[ 1]  = (loopP mod 10) + 48
      ' Uart1_Write_Text(file_contents)
    Mmc_Fat_Write(file_contents, 42)   ' write data to the assigned file
  next loopP
end sub '~

'M-M-M--------- Creates many new files and writes data to them
sub procedure M_Create_Multiple_Files
  'for loop2 = "B" to "Z"
  for loop2 = "B" to "Z"
    Uart1_Write_Char(loop2)                 ' signal the progress
    filename[ 7]  = loop2                   ' set filename
    Mmc_Fat_Assign(filename, 0xA0)     ' find existing file or create a new one
    Mmc_Fat_Rewrite                    ' To clear file and start with new data
    for loopP = 1 to 44
      file_contents[ 0]  = (loopP div 10) + 48
      file_contents[ 1]  = (loopP mod 10) + 48
      Mmc_Fat_Write(file_contents, 42)      ' write data to the assigned file
    next loopP
  next loop2
end sub '~

'M-M-M--------- Opens an existing file and rewrites it
sub procedure M_Open_File_Rewrite
  filename[ 7]  = "C"
  Mmc_Fat_Assign(filename, 0)
  Mmc_Fat_Rewrite
  for loopP = 1 to 55
    file_contents[ 0]  = loopP div 10 + 64
    file_contents[ 1]  = loopP mod 10 + 64
    Mmc_Fat_Write(file_contents, 42)       ' write data to the assigned file
  next loopP
end sub '~


' continues ...
```

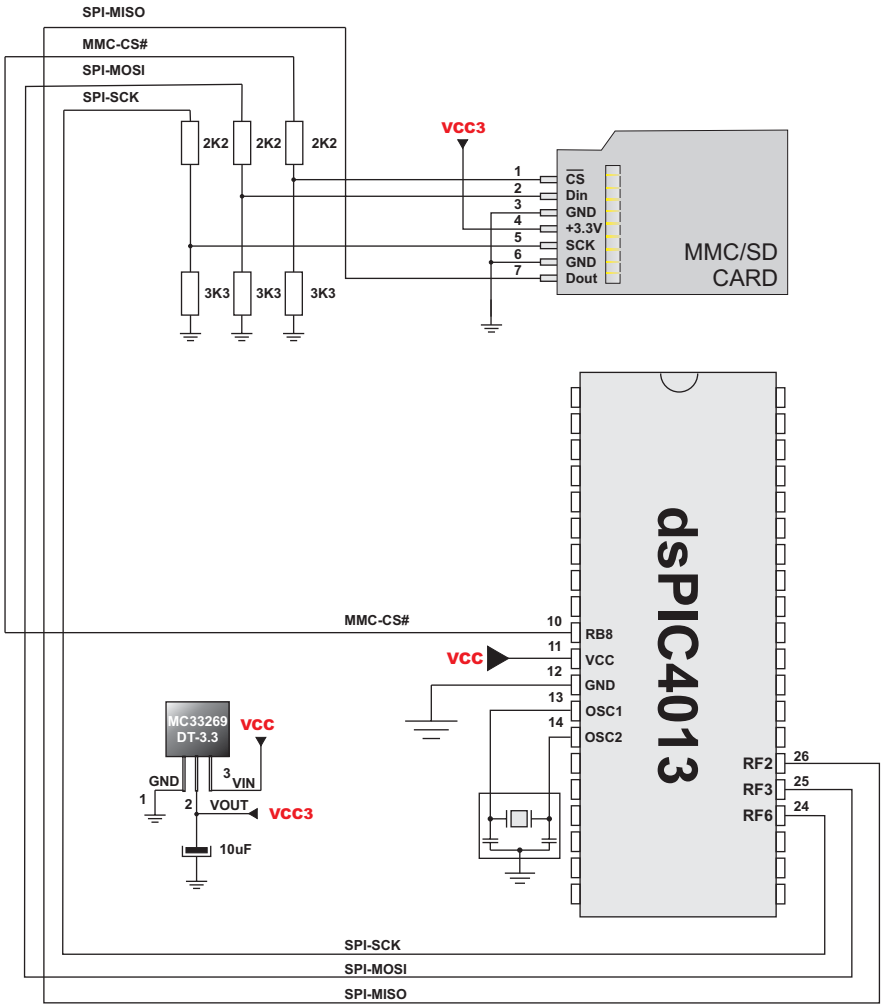
```
' continued ...

'M-M-M--------- Opens an existing file and appends data to it
'                (and alters the date/time stamp)
sub procedure M_Open_File_Append
  filename[ 7]  = "B"
  Mmc_Fat_Assign(filename, 0)
  Mmc_Fat_Set_File_Date(2005,6,21,10,35,0)
  Mmc_Fat_Append                                    ' Prepare file for append
  Mmc_Fat_Write(" for mikroElektronika 2005\n", 27)' Write data to assigned file
end sub '~

'M-M-M--------- Opens an existing file, reads data from it and puts it to USART
sub procedure M_Open_File_Read
  filename[ 7]  = "B"
  Mmc_Fat_Assign(filename, 0)
  Mmc_Fat_Reset(size)           ' To read file, procedure returns size of file
  for i = 1 to size
     ' Mmc_Fat_Read(caracter)
    Mmc_Fat_Read(caracter)
    Uart1_Write_Char(caracter)   ' Write data to USART
  next i
end sub '~

'M-M-M--------- Deletes a file. If file doesn't exist, it will first be created
'                and then deleted.
sub procedure M_Delete_File
  filename[ 7]  = "F"
  Mmc_Fat_Assign(filename, 0)
  Mmc_Fat_Delete
end sub '~



' continues ...
```

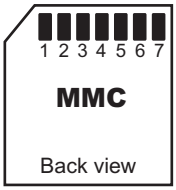
```
' continued ...

'M-M-M--------- Tests whether file exists, and if so sends its creation date
'                and file size via USART
sub procedure M_Test_File_Exist(dim fLetter as byte)
dim
  fsize as longint
  year as word
  monthH, day, hourR, minuteE as byte
  outstr as byte[ 12]

  filename[ 7]  = fLetter
  if (Mmc_Fat_Assign(filename, 0) <> 0) then
    '--- file has been found - get its date
    Mmc_Fat_Get_File_Date(year, monthH, day, hourR, minuteE)
    WordToStr(year, outstr)
    I_Write_Str(outstr)
    WordToStr(monthH, outstr)
    I_Write_Str(outstr)
    WordToStr(day, outstr)
    I_Write_Str(outstr)
    WordToStr(hourR, outstr)
    I_Write_Str(outstr)
    WordToStr(minuteE, outstr)
    I_Write_Str(outstr)
    '--- get file size
    fsize = Mmc_Fat_Get_File_Size
    LongIntToStr(longint(fsize), outstr)
    I_Write_Str(outstr)
  else
    '--- file was not found - signal it
    Uart1_Write_Char(0x55)
    Delay_ms(1000)
    Uart1_Write_Char(0x55)
  end if
end sub '~

' continues ...
```

```
' continued ...

sub procedure M_Create_Swap_File
  dim i as word
    for i=0 to 511
      Buffer[ i]  = i
    next i

    size = Mmc_Fat_Get_Swap_File(5000, "mikroE.txt", 0x20)
    ' see help on this function for details

    if (size <> 0) then
      LongIntToStr(size, fat_txt)
      Uart1_Write_Text(fat_txt)

      for i=0 to 4999
        Mmc_Write_Sector(size, Buffer)
        Inc(size)
        Uart1_Write_Char(".")
      next i
    end if

main:
      '--- Main. Uncomment the function(s) to test the desired operation(s)
      Glob_Init
      '--- prepare PORTD for signalling
      PORTD = 0
      TRISD = 0
      ADPCFG = 0xFFFF
      '--- set up USART for the file read
      Uart1_Init(19200)
      U1MODE.10 = 1 ' clear the way for the SPI
      '--- init the FAT library
      Spi1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1,
_SPI_PRESCALE_PRI_64, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE,
_SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE)

      '--- (primary prescale = 64:1), secondary prescale = 1:1
      ' use fat16 quick format instead of init routine if a format is needed

      if (Mmc_Fat_Init(PORTF, 0) = 0) then ' EasydsPIC4 board
        Spi1_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1,
_SPI_PRESCALE_PRI_4, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE,
_SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE)

        '--- Test start
        PORTD = 0x0005

' continues ...
```

```
' continued ...

        '--- Test routines. Uncomment them one-by-one to test certain features
        M_Create_New_File
        M_Create_Multiple_Files
        Uart1_Write_Char(10)

        M_Open_File_Rewrite
        M_Open_File_Append
        M_Delete_File
        M_Create_Swap_File
        Uart1_Write_Char(10)

        M_Open_File_Read
        Uart1_Write_Char(10)
        M_Test_File_Exist("F")          ' this file will not exist here
        Uart1_Write_Char(10)
        M_Test_File_Exist("B")          ' this file will exist here
        Uart1_Write_Char(10)
      else
        I_Write_Str(fat_txt)
      end if
      '--- Test termination
      PORTD = 0xFFFF
end.
```

Hardware Connection





OneWire Library

The OneWire library provides routines for communication via the Dallas OneWire protocol, e.g. with DS18x20 digital thermometer. OneWire is a Master/Slave protocol, and all communication cabling required is a single wire. OneWire enabled devices should have open collector drivers (with single pull-up resistor) on the shared data line.

Slave devices on the OneWire bus can even get their power supply from data line. For detailed schematic see device datasheet.

Some basic characteristics of this protocol are:

- single master system,
- low cost,
- low transfer rates (up to 16 kbps),
- fairly long distances (up to 300 meters),
- small data transfer packages.

Each OneWire device has also a unique 64-bit registration number (8-bit device type, 48-bit serial number and 8-bit CRC), so multiple slaves can co-exist on the same bus.

Note: Oscillator frequency *Fosc* needs to be at least 4MHz in order to use the routines with Dallas digital thermometers.

Library Routines

```
Ow_Reset
Ow_Read
Ow_Write
```

Ow_Reset

Prototype	`sub function Ow_Reset(dim byref port as word, dim pin as word) as word`
Returns	- `0` if the device is present - `1` if the device is not present
Description	Issues OneWire reset signal for DS18x20. Parameters : - `port`: OneWire bus port - `pin`: OneWire bus pin
Requires	Devices compliant with the Dallas OneWire protocol.
Example	`' Issue Reset signal on One-Wire Bus connected to pin RF6` `Ow_Reset(PORTF, 6)`

Ow_Read

Prototype	`sub function Ow_Read(dim byref port as word, dim pin as word) as byte`
Returns	Data read from an external device over the OneWire bus.
Description	Reads one byte of data via the OneWire bus. Parameters : - `port`: OneWire bus port - `pin`: OneWire bus pin
Requires	Devices compliant with the Dallas OneWire protocol.

Example	`' Read a byte from the One-Wire Bus connected to pin RF6`
	`dim read_data as byte`
	`...`
	`read_data = Ow_Read(PORTF, 6)`

Ow_Write

Prototype	`sub procedure Ow_Write(dim byref port as word, dim pin, par as word)`
Description	Writes one byte of data via the OneWire bus. Parameters : - `port`: OneWire bus port - `pin`: OneWire bus pin - `par`: data to be written
Requires	Devices compliant with the Dallas OneWire protocol.
Example	`' Send a byte to the One-Wire Bus connected to pin RF6` `Ow_Write(PORTF, 6, 0xCC)`

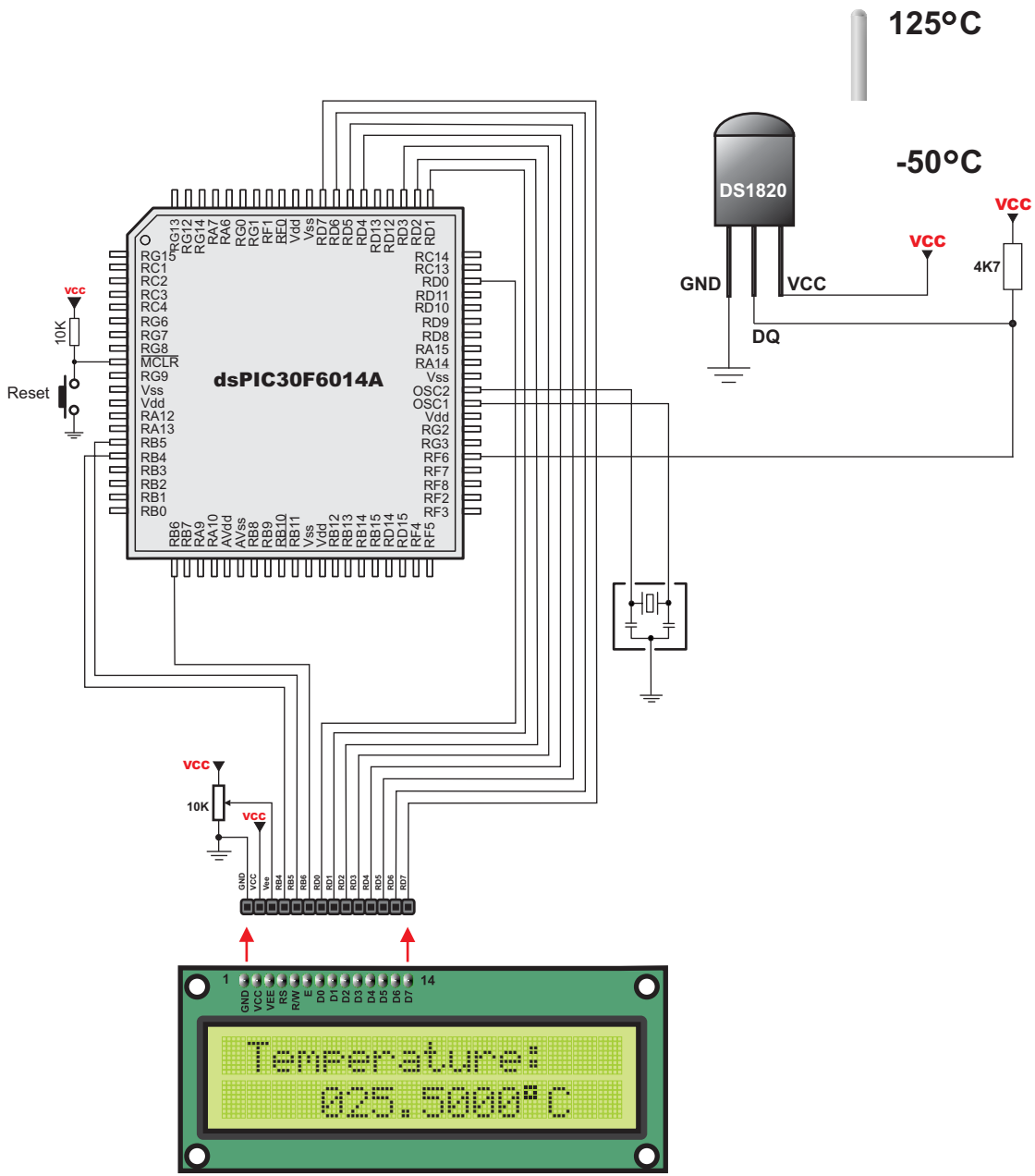
Library Example

This example demonstrates one-wire communication with temperature sensor DS1820 connected to RF6 pin. After reset, dsPIC obtaines temperature from sensor and prints it on LCD.

```
program OneWire

' Set TEMP_RESOLUTION to the corresponding resolution of your DS18x20 sensoras
'  18S20as 9  (default setting can be 9,10,11,or 12)
'  18B20as 12
const TEMP_RESOLUTION as byte = 12

dim
  j as byte
  temp as word
  text as string[ 8]              ' "000.0000"

sub procedure Read_Temperature()
  Ow_Reset(PORTF,6)              ' Onewire reset signal
  Ow_Write(PORTF,6,0xCC)         ' Issue command SKIP_ROM
  Ow_Write(PORTF,6,0x44)         ' Issue command CONVERT_T
  Delay_us(120)

  Ow_Reset(PORTF,6)
  Ow_Write(PORTF,6,0xCC)         ' Issue command SKIP_ROM
  Ow_Write(PORTF,6,0xBE)         ' Issue command READ_SCRATCHPAD
  Delay_ms(400)

  j    = OW_Read(PORTF,6)       ' Get temperature LSB
  temp = OW_Read(PORTF,6)       ' Get temperature MSB
  temp = temp << 8
  temp = temp + j               ' Form the 2byte variable
end sub

sub procedure Write_Temp(dim temp2write as word)
  const
    RES_SHIFT as byte = TEMP_RESOLUTION - 8
  dim
    temp_whole as byte
    temp_fraction as word

  if ((temp and 0x8000) > 0) then
     text[ 0]  = "-"
     temp2write = (not temp2write) + 1
  end if

' continues...
```

```
' continued...

  temp_whole = temp2write >> RES_SHIFT
  if (temp_whole div 100) then
     text[ 0]  = temp_whole div 100   + 48
  end if

  text[ 1]  =(temp_whole div 10) mod 10 + 48
  text[ 2]  = temp_whole mod 10 + 48

  temp_fraction = temp2write << (4-RES_SHIFT)
  temp_fraction = temp_fraction and 0x000F
  temp_fraction = temp_fraction * 625
  text[ 4]  =  temp_fraction div 1000 + 48
  text[ 5]  = (temp_fraction div 100) mod 10 + 48
  text[ 6]  = (temp_fraction div 10)  mod 10 + 48
  text[ 7]  =  temp_fraction mod 10 + 48

  Lcd_Out(2,1,text)
end sub


' ----Main----
main:

  text    = "000.0000"
  ADPCFG = 0xFFFF

  Lcd_Init_dsPICPro3()
  Lcd_Cmd(LCD_CLEAR)
  Lcd_Cmd(LCD_CURSOR_OFF)
  Lcd_Out(1,1,"temperature")

  '--- main loop
  while TRUE
    Read_Temperature()
    Write_Temp(temp)
    Delay_ms(1000)
  wend
end.
```

Hardware Connection



LCD 2X16

Port Expander Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for communication with the Microchip's Port Expander MCP23S17 via SPI interface. Connections of the dsPIC30/33 and PIC24 MCU and MCP23S17 is given on the schematic at the bottom of this page.

Note: Library uses the SPI module for communication. The user must initialize the appropriate SPI module before using the Port Expander Library. For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the Spi_Set_Active() function. See the Spi Library functions.

Note: Library does not use Port Expander interrupts.

Library Routines

```
Expander_Init
Expander_Read_Byte
Expander_Write_Byte
Expander_Read_PortA
Expander_Read_PortB
Expander_Read_PortAB
Expander_Write_PortA
Expander_Write_PortB
Expander_Write_PortAB
Expander_Set_DirectionPortA
Expander_Set_DirectionPortB
Expander_Set_DirectionPortAB
Expander_Set_PullUpsPortA
Expander_Set_PullUpsPortB
Expander_Set_PullUpsPortAB
```

Expander_Init

Prototype	`sub procedure Expander_Init(dim ModuleAddress as byte, dim byref RstPort as word, dim RstPin as word, dim byref CSPort as word, dim CSPin as word)`
Description	Initializes Port Expander using SPI communication. Port Expander module settings : - hardware addressing enabled - automatic address pointer incrementing disabled (byte mode) - BANK_0 register adressing - slew rate enabled Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `rstport`: Port Expander's reset signal port address - `rstpin`: Port Expander's reset signal pin - `csport`: Port Expander's chip select signal port address - `cspin`: Port Expander's chip select signal pin
Requires	The SPI module needs to be initialized. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.

Example	`' initialize the SPI1 module and Port Expander`
	`Spi1_Init()`
	`Expander_Init(0, PORTF, 0, PORTF, 1)`

Expander_Read_Byte

Prototype	`sub function Expander_Read_Byte(dim ModuleAddress, RegAddress as byte) as byte`
Returns	Byte read.
Description	The function reads byte from Port Expander. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `RegAddress`: Port Expander's internal register address
Requires	Port Expander must be initialized. See Expander_Init.

Example	`' Read a byte from Port Expander's register`
	`dim read_data as byte`
	`...`
	`read_data = Expander_Read_Byte(0,1)`

Expander_Write_Byte

Prototype	`sub procedure Expander_Write_Byte(dim ModuleAddress, RegAddress, Data as byte)`
Returns	Nothing.
Description	Routine writes a byte to Port Expander. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `RegAddress`: Port Expander's internal register address - `Data`: data to be written
Requires	Port Expander must be initialized. See Expander_Init.
Example	`' Write a byte to the Port Expander's register` `Expander_Write_Byte(0,1,$FF)`

Expander_Read_PortA

Prototype	`sub function Expander_Read_PortA(dim ModuleAddress as byte) as byte`
Returns	Byte read.
Description	The function reads byte from Port Expander's PortA. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA should be configured as input. See the Expander_Set_DirectionPortA and Expander_Set_DirectionPortAB routines.

Example	`' Read a byte from Port Expander's PORTA`
	`dim read_data as byte`
	`...`
	`Expander_Set_DirectionPortA(0,$FF)`
	`' set expander's porta to be input`
	`...`
	`read_data = Expander_Read_PortA(0)`

Expander_Read_PortB

Prototype	`sub function Expander_Read_PortB(dim ModuleAddress as byte) as byte`
Returns	Byte read.
Description	The function reads byte from Port Expander's PortB. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortB should be configured as input. See the Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	`' Read a byte from Port Expander's PORTB`
	`dim read_data as byte`
	`...`
	`Expander_Set_DirectionPortB(0,$FF)`
	`' set expander's portb to be input`
	`...`
	`read_data = Expander_Read_PortB(0)`

Expander_Read_PortAB

Prototype	`sub function Expander_Read_PortAB(dim ModuleAddress as byte) as word`
Returns	Word read.
Description	The function reads word from Port Expander's ports. PortA readings are in the higher byte of the result. PortB readings are in the lower byte of the result. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA and PortB should be configured as inputs. See the Expander_Set_DirectionPortA, Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	`' Read a byte from Port Expander's PORTA and PORTB`
	`dim read_data as word`
	`...`
	`Expander_Set_DirectionPortAB(0,$FFFF)`
	`' set expander's porta and portb to be input`
	`...`
	`read_data s= Expander_Read_PortAB(0)`

Expander_Write_PortA

Prototype	`sub procedure Expander_Write_PortA(dim ModuleAddress, Data as byte)`
Returns	Nothing.
Description	The function writes byte to Port Expander's PortA. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA should be configured as output. See the Expander_Set_DirectionPortA and Expander_Set_DirectionPortAB routines.

Example	`' Write a byte to Port Expander's PORTA`
	`...`
	`Expander_Set_DirectionPortA(0,$00)`
	`' set expander's porta to be output`
	`...`
	`Expander_Write_PortA(0, $AA)`

Expander_Write_PortB

Prototype	`sub procedure Expander_Write_PortB(dim ModuleAddress, Data as byte)`
Returns	Nothing.
Description	The function writes byte to Port Expander's PortB. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortB should be configured as output. See the Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	`' Write a byte to Port Expander's PORTB`
	`...`
	`Expander_Set_DirectionPortB(0,$00)`
	`' set expander's portb to be output`
	`...`
	`Expander_Write_PortB(0,$55)`

Expander_Write_PortAB

Prototype	`sub procedure Expander_Write_PortAB(dim ModuleAddress as byte, dim Data as word)`
Returns	Nothing.
Description	The function writes word to Port Expander's ports. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written. Data to be written to PortA are passed in Data's higher byte. Data to be written to PortB are passed in Data's lower byte
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA and PortB should be configured as outputs. See the Expander_Set_DirectionPortA, Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	`' Write a byte to Port Expander's PORTA and PORTB`
	`...`
	`Expander_Set_DirectionPortAB(0, $0000)`
	`' set expander's porta and portb to be output`
	`...`
	`Expander_Write_PortAB(0, $AA55)`

Expander_Set_DirectionPortA

Prototype	`sub procedure Expander_Set_DirectionPortA(dim ModuleAddress, Data as byte)`
Description	The function sets Port Expander's PortA direction. Parameters : - ModuleAddress: Port Expander hardware address, see schematic at the bottom of this page - Data: data to be written to the PortA direction register. Each bit corresponds to the appropriate pin of the PortA register. Set bit configures corresponding pin as input. Cleared bit configures corresponding pin as output.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`' Set Port Expander's PORTA to be output` `Expander_Set_DirectionPortA(0,$00)`

Expander_Set_DirectionPortB

Prototype	`sub procedure Expander_Set_DirectionPortB(dim ModuleAddress, Data as byte)`
Description	The function sets Port Expander's PortB direction. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written to the PortB direction register. Each bit corresponds to the appropriate pin of the PortB register. Set bit configures corresponding pin as input. Cleared bit configures corresponding pin as output.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`' Set Port Expander's PORTB to be input` `Expander_Set_DirectionPortB(0,$FF)`

Expander_Set_DirectionPortAB

Prototype	`sub procedure Expander_Set_DirectionPortAB(dim ModuleAddress, Direction as word)`
Description	The function sets Port Expander's PortA and PortB direction. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Direction`: data to be written to direction registers. Data to be written to the PortA direction register are passed in Direction's higher byte. Data to be written to the PortB direction register are passed in Direction's lower byte. Each bit corresponds to the appropriate pin of the PortA/PortB register. Set bit configures corresponding pin as input. Cleared bit configures corresponding pin as output.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`' Set Port Expander's PORTA to be output and PORTB to be input` `Expander_Set_DirectionPortAB(0,$00FF)`

Expander_Set_PullUpsPortA

Prototype	`sub procedure Expander_Set_PullUpsPortA(dim ModuleAddress, Data as byte)`
Description	The function sets Port Expander's PortA pull up/down resistors. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data for choosing pull up/down resistors configuration. Each bit corresponds to the appropriate pin of the PortA register. Set bit enables pull-up for corresponding pin.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`' Set Port Expander's PORTA pull-up resistors` `Expander_Set_PullUpsPortA(0, $FF)`

Expander_Set_PullUpsPortB

Prototype	`sub procedure Expander_Set_PullUpsPortB(dim ModuleAddress, Data as byte)`
Description	The function sets Port Expander's PortB pull up/down resistors. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data for choosing pull up/down resistors configuration. Each bit corresponds to the appropriate pin of the PortB register. Set bit enables pull-up for corresponding pin.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`' Set Port Expander's PORTB pull-up resistors` `Expander_Set_PullUpsPortB(0, $FF)`

Expander_Set_PullUpsPortAB

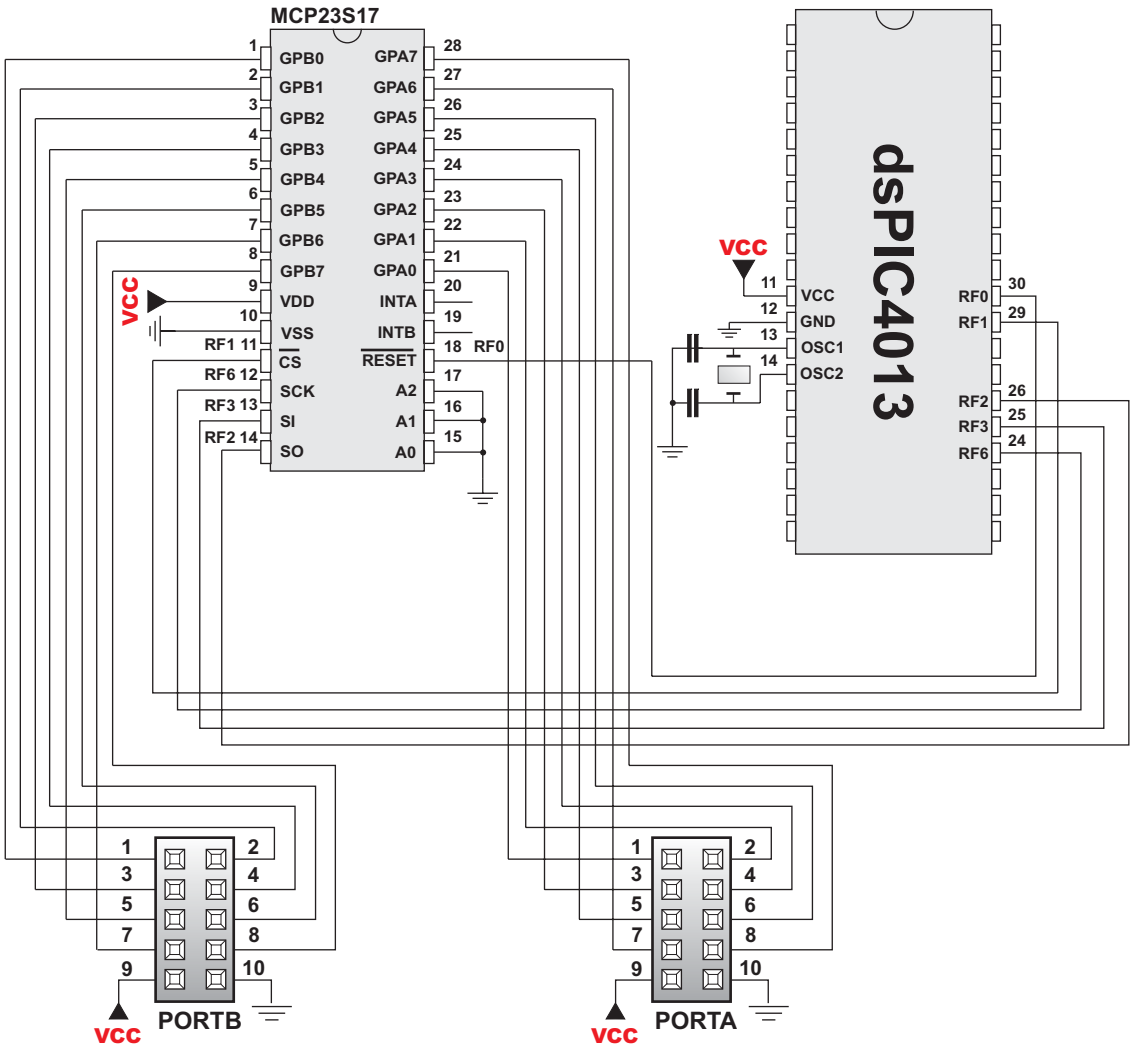
Prototype	`sub procedure Expander_Set_PullUpsPortAB(dim ModuleAddress as byte, dim PullUps as word)`
Description	The function sets Port Expander's PortA and PortB pull up/down resistors. Parameters : - ModuleAddress: Port Expander hardware address, see schematic at the bottom of this page - PullUps: data for choosing pull up/down resistors configuration. PortA pull up/down resistors configuration is passed in PullUps's higher byte. PortB pull up/down resistors configuration is passed in PullUps's lower byte. Each bit corresponds to the appropriate pin of the PortA/PortB register. Set bit enables pull-up for corresponding pin.
Requires	Port Expander must be initialized. See Expander_Init.
Example	`' Set Port Expander's PORTA and PORTB pull-up resistors` `Expander_Set_PullUpsPortAB(0, $FFFF)`

Library Example

The example demonstrates how to communicate with Port Expander MCP23S17. Note that Port Expander pins A2 A1 A0 are connected to GND so Port Expander Hardware Address is 0.

```
program PortExpander

dim i as byte

main:
  ADPCFG = 0xFFFF                                  ' all pins digital
  TRISB  = 0x00                                    ' PORTB output
  PORTB  = 0xFF

  ' for faster spi communication use Spi_Init_Advanced routine
  SPI_init                                         ' initialize SPI
  Delay_ms(200)
  Expander_Init(0, PORTF, 0, PORTB, 8)        ' initialize port expander
  ' on EASYdsPIC4 board RST pin is connected to Vcc
  ' any port/pin can be passed as rstport and rstpin parameters
  ' this pin is used only in Expander_Init function.
  ' This pin can be reassigned later in the code.

  Expander_Set_DirectionPortA(0, 0)      ' set expander's porta to be output

  Expander_Set_DirectionPortB(0, 0xFF)   ' set expander's porta to be input
  Expander_Set_PullUpsPortB(0, 0xFF)
  ' set pull ups to all of the expander's portb pins

  i = 0
  while true
    Expander_Write_PortA(0, i)  ' write i to expander's porta
    i = i + 1
    PORTB = Expander_Read_PortB(0)
    ' read expander's portb and write it to dsPIC's LATB
    Delay_ms(20)
  wend
end.
```

Hardware Connection



PS/2 Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for communication with the common PS/2 keyboard.

Note: The library does not utilize interrupts for data retrieval, and requires the oscillator clock to be at least 6MHz.

Note: The pins to which a PS/2 keyboard is attached should be connected to the pull-up resistors.

Note: Although PS/2 is a two-way communication bus, this library does not provide MCU-to-keyboard communication; e.g. pressing the Caps Lock key will not turn on the Caps Lock LED.

Library Routines

```
Ps2_Init
Ps2_Config
Ps2_Key_Read
```

Ps2_Init

Prototype	`sub procedure Ps2_Init(dim byref port as word)`
Description	Initializes MCU for work with the PS/2 keyboard with default pin settings (pin 13 is connected to Data line and pin 14 is connected to Clock line). Parameters : - `port`: MCU's port initialized for communication with keyboard
Requires	Both Data and Clock lines need to be in the pull-up mode.

Example	`' Initialize PORTC (with default pin settings)`
	`' for work with the PS/2 keyboard`
	`Ps2_Init(PORTC)`

Ps2_Config

Prototype	`sub procedure Ps2_Config(dim byref port as word, dim clkpin as word, dim datapin as word)`
Description	Initializes the MCU for work with the PS/2 keyboard with custom pin settings. Parameters : - `port`: MCU's port initialized for communication with keyboard - `clkpin`: clock signal pin - `datapin`: data signal pin
Requires	Both Data and Clock lines need to be in the pull-up mode.

Example	`' Initialize PORTB for work with the PS/2 keyboard.`
	`' Pin RB2 is connected to Clock signal.`
	`' Pin RB3 is connected to Data signal.`
	
	`Ps2_Config(PORTB, 2, 3)`

Ps2_Key_Read

Prototype	`sub function Ps2_Key_Read(dim byref value as word, dim byref special as word, dim byref pressed as word) as word`
Returns	- `1` if reading of a key from the keyboard was successful - `0` if no key was pressed
Description	The function retrieves information on key pressed. Parameters : - `value`: holds the value of the key pressed. For characters, numerals, punctuation marks, and space value will store the appropriate ASCII code. Routine "recognizes" the function of Shift and Caps Lock, and behaves appropriately. For special function keys see the Special Function Keys Table. - `special`: is a flag for special function keys (F1, Enter, Esc, etc). If key pressed is one of them, special will be set to 1, otherwise it will be set to 0. - `pressed`: is set to 1 if the key is pressed, and 0 if it is released.
Requires	PS/2 keyboard needs to be initialized. See the Ps2_Init and Ps2_Config routines.

Example	`dim value, special, pressed as word`
	`...`
	`' Press Enter to continue:`
	`do {`
	`  if (Ps2_Key_Read(value, special, pressed)) then`
	`  if ((value = 13) and (special = 1)) then`
	`  break`
	`  end if`
	`  end if`
	`loop until (0=1)`

Special Function Keys

Key	Value returned
F1	1
F2	2
F3	3
F4	4
F5	5
F6	6
F7	7
F8	8
F9	9
F10	10
F11	11
F12	12
ENTER	13
PAGE UP	14
PAGE DOWN	15
BACKSPACE	16
INSERT	17

Key	Value returned
DELETE	18
WINDOWS	19
CTRL	20
SHIFT	21
ALT	22
PRINT SCREEN	23
PAUSE	24
CAPS LOCK	25
END	26
HOME	27
SCROLL LOCK	28
NUM LOCK	29
LEFT ARROW	30
RIGHT ARROW	31
UP ARROW	32
DOWN ARROW	33
ESCAPE	34
TAB	35

Library Example

The following example reads values of keys pressed on PS/2 keyboard and sends them via UART1.

```
program ps2_test

dim
 keydata, special, down  as word

main:
  ADPCFG=$FFFF
  keydata = 0
  special = 0
  down    = 0

  Ps2_Config(PORTC, 14, 13)
  Uart1_Init(9600)

  Uart1_Write_Text("You can type now!")  ' Ready
  while TRUE
    if Ps2_Key_Read(keydata, special, down) <> 0 then
      if ((down<>0) and (keydata = 16)) <> 0 then
        Uart1_Write_Char(0x08)
      else
        if ((down<>0) and (keydata = 13)) <> 0 then
          ' Enter
          Uart1_Write_Char(13)
          Uart1_Write_Char(10)
        else
          if ((down<>0) and (special=0) and (keydata<>0)) <> 0 then
            Uart1_Write_Char(keydata)
          end if
        end if
      end if
    end if
    Delay_ms(1)  ' debounce
  wend
end.
```

PWM Library

The CCP module is available with a number of dsPIC30/33 and PIC24 MCUs. mikroBasic for dsPIC30/33 and PIC24 provides a library which simplifies using of the PWM HW Module.

Note: PWM module uses either Timer2 or Timer3 module.

Library Routines

```
Pwm_Init
Pwm_Set_Duty
Pwm_Start
Pwm_Stop
```

Pwm_Init

Prototype	`sub function Pwm_Init(dim freq_hz as longint, dim enable_channel_x, timer_prescale, use_timer_x as word) as word`
Returns	- `0xFFFF` - if timer settings are not valid - Calculated timer period, otherwise
Description	Initializes the PWM module with duty ratio 0. Parameters : - `freq_hz`: PWM frequency in Hz (refer to device datasheet for correct values in respect with Fosc) - `enable_channel_x`: number of PWM channel to be initialized. Refer to MCU's datasheet for available PWM channels - `timer_prescale`: timer prescaler parameter. Valid values: 1, 8, 64, and 256 - `use_timer_x`: timer to be used with the PWM module. Valid values: 2 (Timer2) and 3 (Timer3) **Note**: number of available PWM channels depends on MCU. Refer to MCU datasheet for details.

Example	`' Initializes the PWM module at 5KHz, channel 1,`
	`' no clock prescale, timer2 :`
	`dim pwm_period1 as word`
	`...`
	`pwm_period1 = Pwm_Init(5000, 1, 0, 2)`

Pwm_Set_Duty

Prototype	`sub procedure Pwm_Set_Duty(dim duty, channel as word)`
Description	The function changes PWM duty ratio. Parameters : - `duty`: PWM duty ratio. Valid values: 0 to timer period returned by the Pwm_Init function. - `channel`: number of PWM channel to change duty to. **Note**: number of available PWM channels depends on MCU. Refer to MCU datasheet for details.
Requires	MCU must have the HW PWM Module. PWM channel must be properly initialized. See the Pwm_Init routine.

Example	`' Set channel 1 duty ratio to 50%:`
	`dim pwm_period1 as word`
	`...`
	`Pwm_Set_Duty(pwm_period1 div 2, 1)`

Pwm_Start

Prototype	`sub procedure Pwm_Start(dim enable_channel_x as byte)`
Description	Starts PWM at requested channel . Parameters : - `enable_channel_x`: number of PWM channel **Note**: number of available PWM channels depends on MCU. Refer to MCU datasheet for details.
Requires	MCU must have the HW PWM Module. PWM channel must be properly configured. See the Pwm_Init and Pwm_Set_Duty routines.
Example	`' start PWM at channel 1` `Pwm_Start(1)`

Pwm_Stop

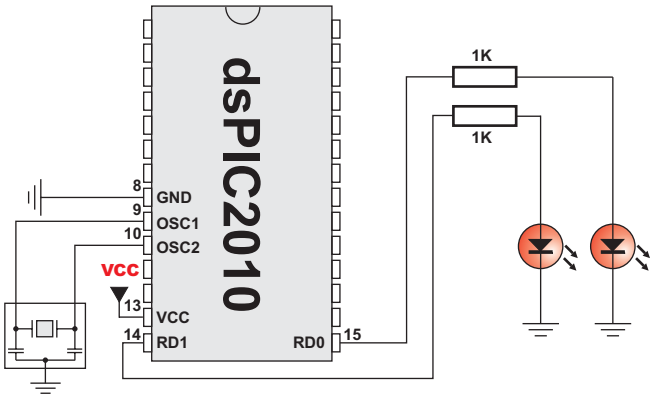
Prototype	`sub procedure Pwm_Stop(dim disable_channel_x as byte)`
Description	Stops PWM at requested channel. Parameters : - `disable_channel_x`: number of PWM channel **Note**: number of available PWM channels depends on MCU. Refer to MCU datasheet for details.
Requires	MCU must have the HW PWM Module.
Example	`' stop PWM at channel 1` `Pwm_Stop(1)`

Library Example

The example changes PWM duty ratio on channels 1 and 2 continuously. If LEDs are connected to channels 1 and 2, a gradual change of emitted light will be noticeable.

```
program PWM_Test

dim pwm_period1, pwm_period2, i1, i2 as word

main:

  i1 = 0
  i2 = 0

  pwm_period1 = Pwm_Init(5000, 1, 1, 2)
  pwm_period2 = Pwm_Init(10000, 2, 1, 3)

  Pwm_Start(1)
  Pwm_Start(2)

  while(true)
    Pwm_Set_Duty(i1, 1)
    Pwm_Set_Duty(i2, 2)
    if (i1 = pwm_period1) then
      i1 = 0
    end if
    if (i2 = pwm_period2) then
      i2 = 0
    end if
    Inc(i1)
    Inc(i2)
    Delay_ms(1)
  wend
end.
```

Hardware Connection



PWM Motor Library

The PWM Motor Control module is available with a number of dsPIC30/33 MCUs. mikroBasic for dsPIC30/33 and PIC24 provides a library which simplifies using the PWM Motor Control module.

Library Routines

```
Pwm_Mc_Init
Pwm_Mc_Set_Duty
Pwm_Mc_Start
Pwm_Mc_Stop
```

Pwm_Mc_Init

Prototype	`sub function Pwm_Mc_Init(dim freq_hz, pair_output_mode, enable_output_x, clock_prescale_output_postscale as word) as word`
Returns	Calculated timer period.
Description	Initializes the Motor Control PWM module with duty ratio 0. The function calculates timer period, writes it to the MCU's PTPER register and returns it as the function result. Parameters : - freq_hz: PWM frequency in Hz (refer to device datasheet for correct values in respect with Fosc) - pair_output_mode: output mode for output pin pairs: 1 = independent, 0 = complementary. if bit_0 is equal to 1 then PWM channels PWM1L and PWM1H will be independent, if bit_1 is equal to 0 then PWM channels PWM2L and PWM2H will be complementary ... - `enable_output_x`: bits <7..0> are enabling corresponding PWM channels [`PWM4H.PWM3H.PWM2H.PWM1H.PWM4L.PWM3L.PWM2L.PWM1L`]. If bit value is equal to 0 then corresponding PWM channel is disabled (pin is standard I/O). If bit value is equal to 1 then corresponding PWM channel is enabled (pin is PWM output). For detalied explanation consult the "Motor Control PWM Module" section in device datasheet - `clock_prescale_output_postscale`: PWM clock prescaler/postscaler settings. Values <0..3> <0..15> correspond to prescaler/postscaler `< 1:1, 1:4, 1:16, 1:64> <1:1, 1:2, ..., 1:16>` **Note**: number of available PWM channels depends on MCU. Refer to MCU datasheet for details.
Requires	The dsPIC30/33 MCU must have the Motor Control PWM module.

Example	`' Initializes the PWM module at 5KHz, complementary pin-pair`
	`' output, output enabled on pins 4l..1l, no clock prescale and no`
	`' clock postscale:`
	`dim duty_50 as word`
	`...`
	`duty_50 = Pwm_Mc_Init(5000, 1, $0F, 0)`

Pwm_Mc_Set_Duty

Prototype	`sub procedure Pwm_Mc_Set_Duty(dim duty, channel as word)`
Description	The function changes PWM duty ratio. Parameters : - `duty`: PWM duty ratio. Valid values: 0 to timer period returned by the Pwm_Mc_Init function. - `channel`: number of PWM channel to change duty to. **Note**: number of available PWM channels depends on MCU. Refer to MCU datasheet for details.
Requires	The dsPIC30/33 MCU must have the Motor Control PWM module. The PWM module needs to be initalized. See the Pwm_Mc_Init function.

Example	`' Set duty ratio to 50% at channel 1:`
	`Pwm_Mc_Init(5000,1,$F,0)`
	`...`
	`Pwm_Mc_Set_Duty(32767, 1)`

Pwm_Mc_Start

Prototype	`sub procedure Pwm_Mc_Start()`
Description	Starts the Motor Control PWM module (channels initialized in the Pwm_Mc_Init function). **Note**: number of available PWM channels depends on MCU. Refer to MCU datasheet for details.
Requires	The dsPIC30/33 MCU must have the Motor Control PWM module. The PWM module needs to be initalized. See the Pwm_Mc_Init function.
Example	`' start the Motor Control PWM module` `Pwm_Mc_Start()`

Pwm_Mc_Stop

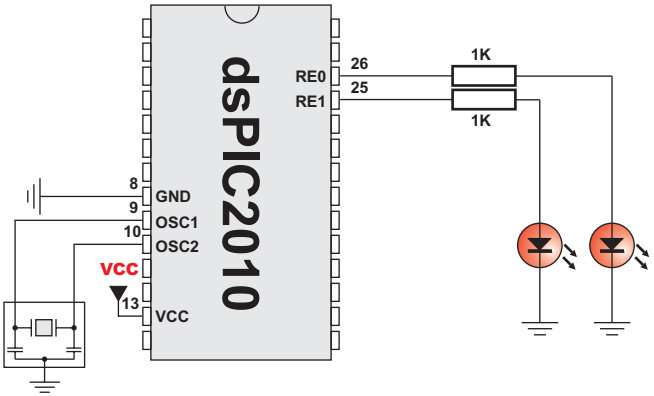
Prototype	`sub procedure Pwm_Mc_Stop()`
Description	Stops the Motor Control PWM module.
Requires	The dsPIC30/33 MCU must have the Motor Control PWM module.
Example	`' stop the Motor Control PWM module` `Pwm_Mc_Stop()`

Library Example

This is a simple example on usage of the PWM library for dsPIC MCUs. Upon reset, PWM module is set up to 50% duty ratio. Then, this ratio is being decreased to zero, and then restored up to approx. 100%, in an infinite loop. The ratio decrease can be observed by turning the PWM output LED on or in digital form on PORTB.

```
program MC_PWM_Test

dim i, duty_50 as word

main:
  ADPCFG = 0xFFFF
  TRISE  = 0
  PORTB  = 0xAAAA
  TRISB  = 0
  Delay_ms(1000)

  duty_50 = Pwm_Mc_Init(5000,1,0x01,0)  ' Pwm_Mc_Init returns 50% of the duty
  i = duty_50
  Pwm_Mc_Set_Duty(i, 1)
  Pwm_Mc_Start()

  while true
    i = i - 1
    Pwm_Mc_Set_Duty(i,1)
    Delay_ms(1)
    if (i = 0) then
      i = duty_50 * 2 - 1              ' Do not allow the overflow
    end if
    PORTB = i
  wend
end.
```

Hardware Connection



RS-485 Library

RS-485 is a multipoint communication which allows multiple devices to be connected to a single bus. mikroBasic for dsPIC30/33 and PIC24 provides a set of library routines for comfortable work with RS485 system using Master/Slave architecture. Master and Slave devices interchange packets of information. Each of these packets contains synchronization bytes, CRC byte, address byte and the data. Each Slave has unique address and receives only packets addressed to it. The Slave can never initiate communication.

It is the user's responsibility to ensure that only one device transmits via 485 bus at a time.

The RS-485 routines require the UART module. Pins of UART need to be attached to RS-485 interface transceiver, such as LTC485 or similar (see schematic at the bottom of this page).

Library constants: START byte value = `150`. STOP byte value = `169`. Address 50 is the broadcast address for all Slaves (packets containing address `50` will be received by all Slaves except the Slaves with addresses `150` and `169`).

Note: The library uses the UART module for communication. The user must initialize the appropriate UART module before using the RS-485 Library. For MCUs with two UART modules it is possible to initialize both of them and then switch by using the `Uart_Set_Active` function. See the Uart Library functions.

Library Routines

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

RS485Master_Init

Prototype	`sub procedure Rs485master_Init(dim byref port as word, dim pin as word)`
Description	Initializes MCU as a Master for RS-485 communication. Parameters : - `port`: RE/DE port address - `pin`: RE/DE pin. This pin is connected to RE/DE input of RS-485 transceiver(see schematic at the bottom of this page). RE/DE signal controls RS-485 transceiver operation mode. Valid values: 1 (for transmitting) and 0 (for receiving)
Requires	UART HW module needs to be initialized. See the Uart1_Init, Uart2_Init and Uart_Init routines.

Example	`' intialize UART module`
	`Uart1_Init(9600)`
	`' intialize mcu as a Master for RS-485 communication`
	`Rs485master_Init(PORTF, 4)`

RS485Master_Receive

Prototype	`sub procedure Rs485master_Receive(dim byref data as byte[ 10] )`
Description	Receives messages from Slaves. Messages are multi-byte, so this routine must be called for each byte received. Parameters : - data: 7 byte buffer for storing received data. Data will be stored in the following manner: `data[ 0..2]` : message content `data[ 3]` : number of message bytes received, 1–3 `data[ 4]` : is set to 255 when message is received `data[ 5]` : is set to 255 if error has occurred `data[ 6]` : address of the Slave which sent the message The function automatically adjusts `data[ 4]` and `data[ 5]` upon every received message. These flags need to be cleared by software.
Requires	MCU must be initialized as a Master for RS-485 communication. See RS485Master_Init.

Example	`dim msg as byte[ 8]`
	`...`
	`RS485Master_Receive(msg)`

RS485Master_Send

Prototype	`sub procedure Rs485master_Send(dim byref data as byte[ 1] , dim datalen as byte, dim address as byte)`
Description	Sends message to Slave(s). Message format can be found at the bottom of this page. Parameters : - `data`: data to be sent - `datalen`: number of bytes for transmition. Valid values: 0 to 3. - `address`: Slave(s) address
Requires	MCU must be initialized as a Master for RS-485 communication. See RS485Master_Init. It is the user's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.

Example	`dim msg as byte[ 8]`
	`...`
	`' send 3 bytes of data to slave with address 0x12`
	`RS485Master_Send(msg, 3, 0x12)`

RS485Slave_Init

Prototype	`sub procedure Rs485slave_Init(dim byref port as word, dim pin as word, dim address as byte)`
Description	Initializes MCU as a Slave for RS-485 communication. Parameters : - `port`: RE/DE port address - `pin`: RE/DE pin. This pin is connected to RE/DE input of RS-485 transceiver (see schematic at the bottom of this page). RE/DE signal controls RS-485 transceiver operation mode. Valid values: 1 (for transmitting) and 0 (for receiving) - `address`: Slave address
Requires	UART HW module needs to be initialized. See Uart_Init.

Example	Initialize MCU as a Slave with address 160:
	
	`' intialize mcu as a Slave for RS-485 communication`
	`' with address 160`
	`Rs485slave_Init(PORTF, 4, 160)`

RS485Slave_Receive

Prototype	`sub procedure Rs485Slave_Receive(dim byref data as byte[ 10] )`
Description	Receives messages from Master. If Slave address and Message address field don't match then the message will be discarded. Messages are multi-byte, so this routine must be called for each byte received. Parameters : - `data`: 6 byte buffer for storing received data. Data will be stored in the following manner: `data[ 0..2]` : message content `data[ 3]` : number of message bytes received, 1–3 `data[ 4]` : is set to 255 when message is received `data[ 5]` : is set to 255 if error has occurred The function automatically adjusts `data[ 4]` and `data[ 5]` upon every received message. These flags need to be cleared by software.
Requires	MCU must be initialized as a Slave for RS-485 communication. See RS485Slave_Init.

Example	`dim msg as byte[ 8]`
	`...`
	`RS485Slave_Read(msg)`

RS485Slave_Send

Prototype	`sub procedure Rs485slave_Send(dim byref data as byte[ 1] , dim datalen as byte)`
Description	Sends message to Master. Message format can be found at the bottom of this page. Parameters : - `data`: data to be sent - `datalen`: number of bytes for transmition. Valid values: 0 to 3.
Requires	MCU must be initialized as a Slave for RS-485 communication. See RS485Slave_Init. It is the user's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.

Example	`dim msg as byte[ 8]`
	`...`
	`' send 2 bytes of data to the master`
	`RS485Slave_Send(msg, 2)`

Message format and CRC calculations

Q: How is CRC checksum calculated on RS485 master side?

```
const START_BYTE as byte = 0x96   ' 10010110
const STOP_BYTE  as byte = 0xA9   ' 10101001

PACKAGE:
--------
START_BYTE 0x96
ADDRESS
DATALEN
[ DATA1]              ' if exists
[ DATA2]              ' if exists
[ DATA3]              ' if exists
CRC
STOP_BYTE   0xA9


DATALEN bits:
------------
bit7 = 1  MASTER SENDS
        0  SLAVE  SENDS
bit6 = 1  ADDRESS WAS XORed with 1, IT WAS EQUAL TO START_BYTE or STOP_BYTE
        0  ADDRESS UNCHANGED
bit5 = 0  FIXED
bit4 = 1  DATA3 (if exists) WAS XORed with 1, IT WAS EQUAL TO START_BYTE or
STOP_BYTE
        0  DATA3 (if exists) UNCHANGED
bit3 = 1  DATA2 (if exists) WAS XORed with 1, IT WAS EQUAL TO START_BYTE or
STOP_BYTE
        0  DATA2 (if exists) UNCHANGED
bit2 = 1  DATA1 (if exists) WAS XORed with 1, IT WAS EQUAL TO START_BYTE or
STOP_BYTE
        0  DATA1 (if exists) UNCHANGED
bit1bit0 = 0 to 3 NUMBER OF DATA BYTES SEND


CRC generation :
----------------
crc_send = datalen  xor address
crc_send = crc_send xor data[ 0]     ' if exists
crc_send = crc_send xor data[ 1]     ' if exists
crc_send = crc_send xor data[ 2]     ' if exists
crc_send = crc_send not crc_send
if ((crc_send = START_BYTE) or (crc_send = STOP_BYTE)) then
   crc_send = crc_send + 1
end if
NOTE: DATALEN<4..0> can not take the START_BYTE<4..0> or STOP_BYTE<4..0> values.
```

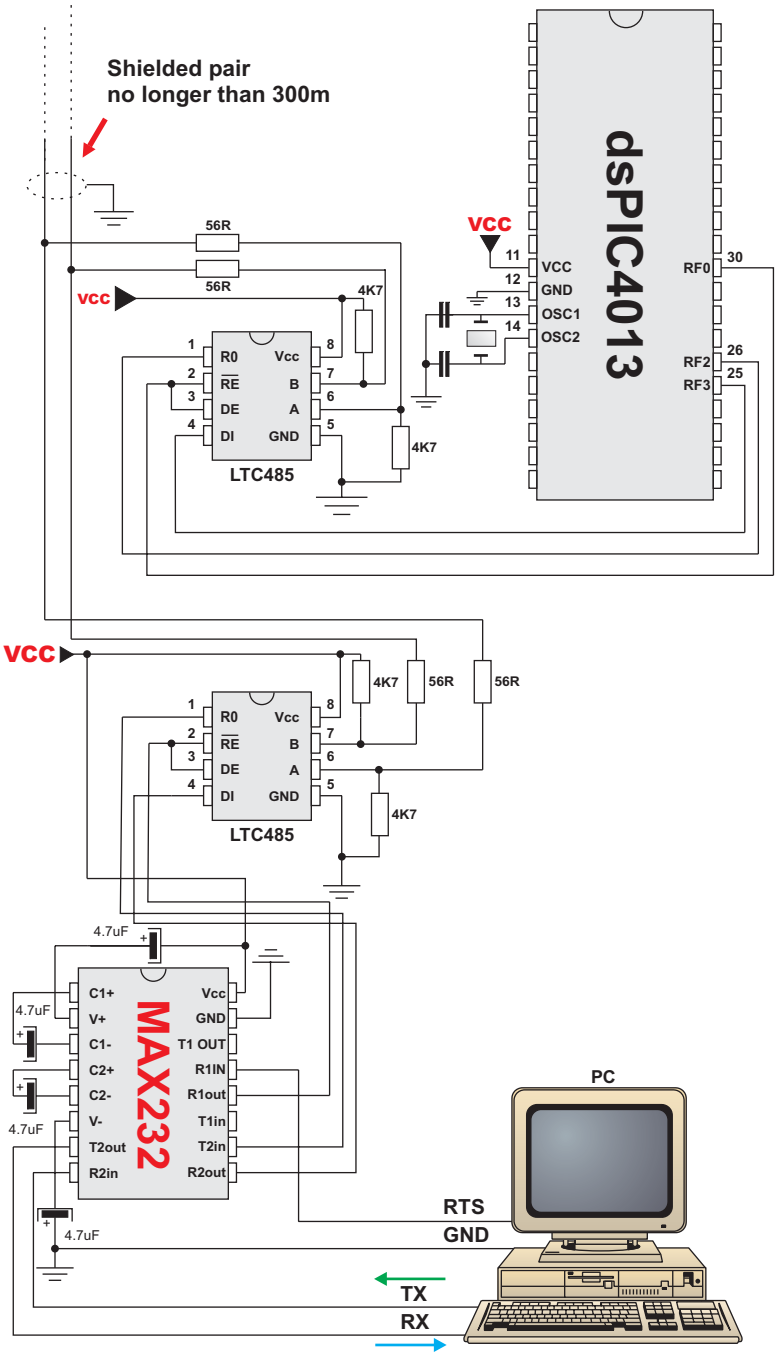
Library Example

The example demonstrates working with the dsPIC as a Master node in RS-485 communication. Master sends message to Slave with address 160 and waits for a response. After the response is received, the first byte of received data is incremented and sent back to the Slave. The received data is displayed on PORTB while error on receiving (0xAA) and number of consecutive unsuccessful retries are displayed on PORTD. Hardware configurations in this example are made for the EasydsPIC3 board and dsPIC30F4013.

```
program rs485Transmiter

dim dat as byte[ 10]                        ' buffer for receving/sending messages
    i,j as integer
    cnt as longint
'-------------- Interrupt routine
sub procedure interrupt_uart() org 0x26
    Rs485master_Receive(dat)
    ' IFS0bits.U1RXIF = 0;
    IFS0.9 = 0                              ' ensure interrupt not pending
end sub

main:
  cnt     = 0
  ADPCFG = 0xFFFF
  PORTD  = 0
  TRISD  = 0
  PORTB  = 0
  TRISB  = 0
  Uart1_Init(9600)
  Rs485master_Init(PORTF, 4)               ' intialize mcu as master
  dat[ 0]  = 0xAa
  dat[ 1]  = 0xF0
  dat[ 2]  = 0x0F
  dat[ 5]  = 0
  dat[ 6]  = 0
  Rs485master_Send(dat, 1, 160)
  dat[ 4]  = 0                             ' ensure that message received flag is 0
  dat[ 5]  = 0                             ' ensure that error flag is 0
  ' U1STAbits.URXISEL = 0;
  U1STA.6 = 0
  ' INTCON1bits.NSTDIS = 1;
  INTCON1.15 = 1                           ' no nesting of interrupts
  'IFS0bits.U1RXIF = 0;
  IFS0.9 = 0                               ' ensure interrupt not pending
  ' IEC0bits.U1RXIE = 1;
  IEC0.9 = 1                               ' enable intterupt

' continues...
```

```
' continued...

  while true
                                        ' upon completed valid message receiving
                                        '   data[4] is set to 255
    cnt = cnt + 1
    if dat[ 5]  <> 0 then               ' if error detected, signal it by
      PORTD = 0xAA                      '   setting PORTD to 0xAA
    end if
    if dat[ 4]  <> 0 then
                                        ' if message received successfully
      cnt     = 0
      dat[ 4]  = 0                      '   clear message received flag
      j = dat[ 3]
      for i = 1 to j
          PORTB = dat[ i-1]             ' show data on PORTB
      next i                            ' increment received dat[0]
      dat[ 0]  = dat[ 0]  + 1           '   and send it to slave
      Delay_ms(10)
      Rs485master_Send(dat, 1, 160)
    end if
    if (cnt > 100000) then
                                        ' if in 100000 poll-cycles the answer

      PORTD = PORTD + 1                 '   was not detected, signal

      cnt = 0                           '   failure of send-message
      Rs485master_Send(dat, 1, 160)
      if(PORTD > 10) then               ' if sending failed 10 times
        PORTD = 0
        Rs485master_Send(dat, 1, 50)    '   send message on broadcast address
      end if
    end if
  wend
end.
```

Hardware Connection



Software I2C Library

mikroBasic for dsPIC30/33 and PIC24 provides routines for implementing Software I²C communication. These routines are hardware independent and can be used with any MCU. The Software I²C library 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 Software I²C.

Note: All I²C Library functions are blocking-call functions (they are waiting for I²C clock line to become logical one).

Library Routines

```
Soft_I2C_Init
Soft_I2C_Start
Soft_I2C_Read
Soft_I2C_Write
Soft_I2C_Stop
```

Note: Every hardware I²C library function has its own counterpart in this software library, except `I2C_Repeated_Start`. `Soft_I2C_Start` is used instead of `I2C_Repeated_Start`.

Soft_I2C_Init

Prototype	`sub procedure Soft_I2C_Init(dim byref portOut as word, dim sda, clk as word)`
Description	Configures the software I²C module. Parameters : - `portOut`: software I²C port address - `sda`: serial data line pin - `clk`: serial clock line pin

Example	`' Initialize Software I2C communication on pins RB1(sda)`
	`' and RB2(clk)`
	`Soft_I2C_Init(PORTB, 1, 2)`

Soft_I2C_Start

Prototype	`sub procedure Soft_I2C_Start()`
Description	Determines if the I²C bus is free and issues START signal.
Requires	Software I²C must be configured before using this function. See the Soft_I2C_Init routine.
Example	`' Issue START signal` `Soft_I2C_Start()`

Soft_I2C_Read

Prototype	`sub function Soft_I2C_Read(dim ack as word) as byte`
Returns	One byte from the Slave.
Description	Reads one byte from the slave. Parameters : - `ack`: acknowledge signal parameter. If the `ack==0` not acknowledge signal will be sent after reading, otherwise the acknowledge signal will be sent.
Requires	Soft I²C must be configured before using this function. See the Soft_I2C_Init routine. Also, START signal needs to be issued in order to use this function. See the Soft_I2C_Start routine.

Example	`dim take as byte`
	`...`
	`' Read data and send the not_acknowledge signal`
	`take = Soft_I2C_Read(0)`

Soft_I2C_Write

Prototype	`sub function Soft_I2C_Write(dim Data as byte) as byte`
Returns	- `0` if there were no errors. - `1` if write collision was detected on the I²C bus.
Description	Sends data byte via the I²C bus. Parameters : - `Data`: data to be sent
Requires	Soft I²C must be configured before using this function. See the Soft_I2C_Init routine. Also, START signal needs to be issued in order to use this function. See the Soft_I2C_Start routine.

Example	`dim data, error as byte`
	`...`
	`error = Soft_I2C_Write(data)`
	`error = Soft_I2C_Write($A3)`

Soft_I2C_Stop

Prototype	`sub procedure Soft_I2C_Stop()`
Description	Issues STOP signal.
Requires	Soft I²C must be configured before using this function. See the Soft_I2C_Init routine.
Example	`' Issue STOP signal` `Soft_I2C_Stop()`

Library Example

The example demonstrates use of the Software I²C Library. The dsPIC30/33 or PIC24 MCU is connected (SCL, SDA pins) to PCF8583 RTC (real-time clock).

```
program Soft_RTC_WriteRead

dim sec, minN, hr, day, mn, year as word
    txt as string[ 15]
    tmp as char[ 5]

sub procedure Set_Time(dim byref sec, minN, hr, day, mn, year as byte)
  ' from real TIME to VARIABLE

  sec  = ((sec div 10) << 4) or (sec mod 10)
  minN = ((minN div 10) << 4) or (minN mod 10)
  hr   = ((hr div 10)  << 4) or (hr mod 10)
  day  = ((day div 10) << 4) or (day mod 10)
  mn   = ((mn div 10) << 4)  or (mn mod 10)

  year = year-2005
  year = year << 6
  day  = day or year
end sub

' Reads time and date information from RTC (PCF8583)
sub procedure Read_Time(dim byref sec, minN, hr, day, mn, year as word)
   year = 0
   Soft_I2C_Start()
   Soft_I2C_Write($A0)
   Soft_I2C_Write(2)
   Soft_I2C_Start()
   Soft_I2C_Write($A1)
   sec = Soft_I2C_Read(1)
   minN = Soft_I2C_Read(1)
   hr  = Soft_I2C_Read(1)
   day = Soft_I2C_Read(1)
   mn  = Soft_I2C_Read(0)
   Soft_I2C_Stop()
end sub

' continues ...
```

```
' continued ...

sub procedure Write_Time_And_Date(dim sec, minN, hr, day, mn as byte)
   Soft_i2C_Start()            ' Issue start signal
   Soft_I2C_Write($A0)         ' Address PCF8583
   Soft_I2C_Write(0)         ' Start from word at address 0 (configuration word)
   Soft_I2C_Write($80)         ' Write $80 to config. (pause counter...)
   Soft_I2C_Write(0)           ' Write 0 to hundreds of seconds word
   Soft_I2C_Write(sec)         ' Write seconds
   Soft_I2C_Write(minN)         ' Write minutes
   Soft_I2C_Write(hr)          ' Write hours
   Soft_I2C_Write(day)         ' Write year/date
   Soft_I2C_Write(mn)          ' Write weekday/month
   Soft_I2C_Stop()             ' issue stop signal

   Soft_i2C_Start()            ' Issue start signal
   Soft_I2C_Write($A0)         ' Address PCF8530
   Soft_I2C_Write(0)           ' Start from word at address 0
   Soft_I2C_Write(0)           ' Write 0 to config. word (enable counting)
   Soft_I2C_Stop()             ' Issue sop signal
end sub

main:

'  Here you can set the date and time to write to RTC

  sec  = 0     ' seconds
  minN = 40    ' minutes
  hr   = 10    ' hours
  day  = 26    ' day
  mn   = 8     ' month
  year = 2006   ' year

  ADPCFG = $FFFF

  Glcd_Init_EasydsPIC4()
  Glcd_Fill(0)

  Soft_I2C_Init(PORTB, 12, 11)
  Glcd_Set_Font(@FontSystem5x8,5,8,32)
  Glcd_Write_Text("Time :",10,0,1)
  Glcd_Write_Text("      :      :"   ,10,1,1)
  Glcd_Write_Text("Date :",10,3,1)
  Glcd_Write_Text("  Software I2C",10,7,1)

  Set_Time(sec, minN, hr, day, mn, year)

  Write_Time_And_Date(sec, minN, hr, day, mn)

' continues ...
```

```
' continued ...

  while true
    Read_Time(sec, minN, hr, day, mn, year)
    sec  = ((sec and $F0)  >> 4)*10 + (sec and $0F)
    minN  = ((minN and $F0)  >> 4)*10 + (minN and $0F)
    hr   = ((hr  and $F0)  >> 4)*10 + (hr  and $0F)
    year = ((day and $C0)) >> 6
    day  = ((day and $30)  >> 4)*10 + (day and $0F)
    mn   = ((mn  and $10)  >> 4)*10 + (mn  and $0F)

    year = year + 2005

    select case mn
      case 1  txt = "Jan"
      case 2  txt = "Feb"
      case 3  txt = "Mar"
      case 4  txt = "Apr"
      case 5  txt = "May"
      case 6  txt = "Jun"
      case 7  txt = "Jul"
      case 8  txt = "Aug"
      case 9  txt = "Sep"
      case 10 txt = "Oct"
      case 11 txt = "Nov"
      case 12 txt = "Dec"
      case else
               txt = "unknown"
    end select

    Glcd_Write_Text(txt, 45, 4, 1)

    WordToStr(sec, tmp)
    Glcd_Write_Text(tmp, 80, 1, 1)

    WordToStr(minN, tmp)
    Glcd_Write_Text(tmp, 45, 1, 1)

    WordToStr(hr, tmp)
    Glcd_Write_Text(tmp, 10, 1, 1)

    WordToStr(day, tmp)
    Glcd_Write_Text(tmp, 10, 4, 1)

    WordToStr(year, tmp)
    Glcd_Write_Text(tmp, 70, 4, 1)

    Delay_ms(200)
  wend
end.
```

Software SPI Library

mikroBasic for dsPIC30/33 and PIC24 provides routines for implementing Software SPI communication. These routines are hardware independent and can be used with any MCU. The Software SPI Library provides easy communication with other devices via SPI: A/D converters, D/A converters, MAX7219, LTC1290, etc.

The library configures SPI to the master mode, clock = 20kHz, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge.

Note: The Software SPI library implements time-based activities, so interrupts need to be disabled when using it.

Library Routines

```
Soft_Spi_Config
Soft_Spi_Init
Soft_Spi_Read
Soft_Spi_Write
```

Soft_Spi_Config

Prototype	`sub procedure Soft_Spi_Config(dim byref port as word, dim SDI, SDO, SCK as word)`
Description	Configures and initializes the software SPI module. Parameters : - `portOut`: software SPI port address - `SDI`: serial data input line pin - `SDO`: serial data output line pin - `SCK`: serial clock line pin

Example	`' Initialize Software SPI communication on pins RB4(SDI),`
	`' RB5(SDO) and RB3(SCK)`
	`Soft_Spi_Config(PORTB, 4, 5, 3)`

Soft_Spi_Init

Prototype	`sub procedure Soft_Spi_Init(dim byref port as word)`
Description	Configures and initializes the software SPI module. Parameters : - `port`: software SPI port address SPI pinout is fixed to : SDI=pin4, SDO=pin5 and SCK=pin3.

Example	`' Initialize Software SPI communication on PORTB with default`
	`' pinout( pin4(SDI), pin5(SDO) and pin3(SCK) )`
	`Soft_Spi_Init(PORTB)`

Soft_Spi_Read

Prototype	`sub function Soft_Spi_Read(dim data as byte) as byte`
Returns	Byte received via the SPI bus.
Description	This routine performs 3 operations simultaneously. It provides clock for the Software SPI bus, reads a byte and sends a byte. Parameters : - `data`: data to be sent.
Requires	Soft SPI must be initialized before using this function. See the Soft_Spi_Config and Soft_Spi_Init routines.

Example	`dim data_read, data_send as byte`
	`...`
	`' Read a byte and assign it to data_read variable`
	`' (data_send byte will be sent via SPI during the Read operation)`
	`data_read = Soft_Spi_Read(data_send)`

Soft_Spi_Write

Prototype	`sub procedure Soft_Spi_Write(dim data as byte)`
Description	This routine sends one byte via the Software SPI bus. Parameters : - `data`: data to be sent.
Requires	Soft SPI must be initialized before using this function. See the Soft_Spi_Config and Soft_Spi_Init routines.
Example	`' Write a byte to the Soft SPI bus` `Soft_Spi_Write($AA)`

Library Example

This code demonstrates using library routines for SPI communication. The project shows how to drive the DAC MCP 4921 (Microchip) via the software SPI library.

```
program Soft_spi_test
const  CS_PIN = 0                          ' DAC CS pin
dim value as word
sub procedure InitMain()
  ADPCFG = 0xFFFF
  TRISF.CS_PIN    = 0                      ' CS pin
  LATF.CS_PIN     = 1                      ' Set CS to inactive
  Soft_Spi_Config(PORTF, 4, 5, 3)
end sub
' DAC increments (0..4095) --> output voltage (0..Vref)
sub procedure DAC_Output(dim valueDAC as word)
  LATF.CS_PIN = 0                                          ' CS enable for DAC
  Soft_SPI_Write(0x30 or ((valueDAC >> 8) and 0x0F))
  ' Prepare hi-byte for transfer
  ' It's a 12-bit number, so only
  '   lower nibble of high byte is used
  Soft_SPI_Write(valueDAC)  ' Prepare lo-byte for transfer
  LATF.CS_PIN     = 1       ' CS disable for DAC
end sub

main:
  InitMain()
  while true
    value = 1
    while value < $FFF
     DAC_Output(value)
     Delay_ms(5)
     value = value+1
    wend
  wend   ' the output in the mid-range
end.
```

Software UART Library

mikroBasic for dsPIC30/33 and PIC24 provides routines for implementing Software UART communication. These routines are hardware independent and can be used with any MCU. The Software UART Library provides easy communication with other devices via the RS232 protocol.

Note: The Software UART library implements time-based activities, so interrupts need to be disabled when using it.

Library Routines

```
Soft_Uart_Init
Soft_Uart_Read
Soft_Uart_Write
```

Soft_Uart_Init

Prototype	`sub procedure Soft_Uart_Init(dim byref port as word, dim rx, tx as word, dim baud_rate as longint, dim inverted as word)`
Description	Configures and initializes the software UART module. Parameters : - `port`: software UART port address - `rx`: receiver pin - `tx`: transmiter pin - `baud_rate`: requested baudrate. Maximum baud rate depends on the MCU's clock and working conditions - `inverted`: if set to non-zero value, indicates inverted logic on output.

Example	`' Initialize Software UART communication on pins RB1(Rx),`
	`' RB2(Tx), at 9600 bps`
	`Soft_Uart_Init(PORTB, 1, 2, 9600, 0)`

Soft_Uart_Read

Prototype	`sub function Soft_Uart_Read(dim byref error as word) as byte`
Returns	Byte received via UART.
Description	The function receives a byte via software UART. This is a blocking function call (waits for the start bit). Parameters : - `error`: error parameter will be zero if the transfer was successful.
Requires	Software UART must be initialized before using this function. See the Soft_Uart_Init routine.

Example	`dim data as byte`
	`  error as word`
	`  ...`
	`  ' wait until data is received`
	`do`
	`  data = Soft_Uart_Read(error)`
	`loop until (error = 0)`

Soft_Uart_Write

Prototype	`sub procedure Soft_Uart_Write(dim data as byte)`
Description	This routine sends one byte via the Software UART bus. Parameters : - `data`: data to be sent.
Requires	Software UART must be initialized before using this function. See the Soft_Uart_Init routine. Be aware that during transmission, software UART is incapable of receiving data – data transfer protocol must be set in such a way to prevent loss of information.

Example	`dim some_byte as byte`
	`...`
	`some_byte = $0A`
	`' Write a byte via Soft Uart`
	`Soft_Uart_Write(some_byte)`

Library Example

The example demonstrates a simple data exchange via software UART. When the MCU receives data, it immediately sends it back. If dsPIC is connected to the PC, you can test the example from the mikroBasic terminal for RS-232 communication, menu item **Tools › USART Terminal**.

```
program Soft_Uart_Example

dim rx, _err as word

main:
  ADPCFG = $FFFF
  Soft_Uart_Init(PORTF, 2, 3, 9600, 0)
  while true
    rx = Soft_Uart_Read(_err)
    Soft_Uart_Write(rx)
  wend
end.
```

Sound Library

mikroBasic for dsPIC30/33 and PIC24 provides a Sound Library to supply users with routines necessary for sound signalization in their applications. Sound generation needs additional hardware, such as piezo-speaker (example of piezo-speaker interface is given on the schematic at the bottom of this page).

Library Routines

```
Sound_Init
Sound_Play
```

Sound_Init

Prototype	`sub procedure Sound_Init(dim byref snd_port, snd_pin as word)`
Description	Configures the appropriate MCU pin for sound generation. Parameters : - `snd_port`: sound output port address - `snd_pin`: sound output pin
Example	`' Initialize the pin RB2 for playing sound` `Sound_Init(PORTB, 2)`

Sound_Play

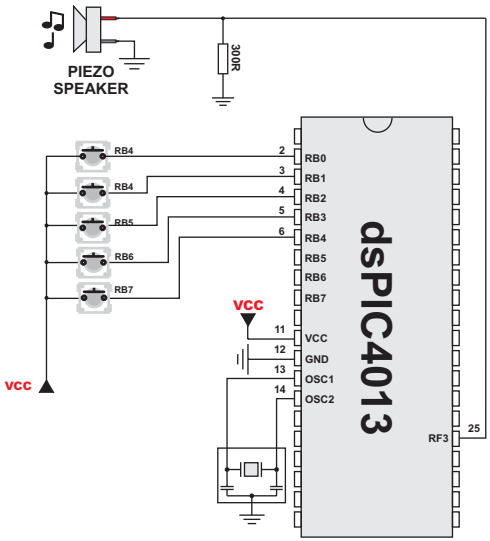
Prototype	`sub procedure Sound_Play(dim freq_in_hz, duration_ms as word)`
Description	Generates the square wave signal on the appropriate pin. Parameters : - `freq_in_hz`: signal frequency in Hertz (Hz) - `duration_ms`: signal duration in miliseconds (ms)
Example	`' Play sound of 1KHz in duration of 100ms` `Sound_Play(1000, 100)`

Library Example

The example is a simple demonstration of how to use sound library for playing tones on a piezo speaker. The code can be used with any MCU that has PORTB. Sound frequencies in this example are generated by reading the buttons on PORTB.

```
program sound
main:
  ' initialize sound lib
  Sound_Init(PORTB, 3);

  ' three beeps
  Sound_Play(1000, 500)
  delay_ms(50)
  Sound_Play(1000, 500)
  delay_ms(50)
  Sound_Play(1000, 500)
  delay_ms(50)
  Sound_Play(1200, 500)    ' frequency is 1200 Hz and duration is 500ms
end.
```

Hardware Connection



SPI Library

The SPI module is available with all dsPIC30/33 and PIC24 MCUs. mikroBasic for dsPIC30/33 and PIC24 provides a library for initializing the Slave mode and initializing and comfortable work with the Master mode. The dsPIC30/33 and PIC24 can easily communicate with other devices via SPI: A/D converters, D/A converters, MAX7219, LTC1290, etc.

Note: For the dsPIC30/33 and PIC24 MCUs with the multiple SPI modules there are SPI1 (supports SPI1 module), SPI2 (supports SPI2 module) and SPI (supports both SPI modules) libraries. Switching between the SPI modules in the SPI library is done by the Spi_Set_Active function (both SPI modules have to be previously initialized).

Library Routines

```
Spi1_Init
Spi1_Init_Advanced
Spi1_Data_Ready
Spi1_Read
Spi1_Write
Spi1_Stop
SPI1_TxBuffer_Empty
```

```
Spi2_Init
Spi2_Init_Advanced
Spi2_Data_Ready
Spi2_Read
Spi2_Write
Spi2_Stop
SPI2_TxBuffer_Empty
```

```
Spi_Init
Spi_Init_Advanced
Spi_Data_Ready
Spi_Read
Spi_Write
Spi_Stop
SPI_TxBuffer_Empty
```

Spi1_Init

Prototype	`sub procedure Spi1_Init()`
Description	Configures and initializes the SPI1 module with default settings. Default settings: - Master mode - 8-bit data mode - secondary prescaler 1:1 - primary prescaler 64:1 - Slave Select disabled - input data sampled in the middle of interval - clock idle state low - Serial output data changes on transition from active clock state to idle clock state
Requires	MCU must have the SPI1 module.
Example	`' Initialize the SPI1 module with default settings` `Spi1_Init()`

Spi1_Init_Advanced

Prototype	`sub procedure Spi1_Init_Advanced(dim master_mode, mode16, sec_prescaler, pri_prescaler, slave_select, data_sample, clock_idle, edge as word)`

Description	Configures and initializes the SPI1 module.
	
	The `master_mode` parameter determines the working mode for SPI1. It can have the following values:
	
	`'--- Master/Slave mode, SPI1CON1<5>`
	`const _SPI_MASTER = $0020, ' Master mode`
	`const _SPI_SLAVE = $0000, ' Slave mode`
	
	The parameter mode16 determines the data length mode, which can be 8-bits (per transmitions cycle) or 16-bits. It is being configured as follows:
	
	`'--- data length select, SPI1CON1<10>`
	`const _SPI_16_BIT = $0400, ' 16-bit mode`
	`const _SPI_8_BIT = $0000, ' 8-bit mode`
	
	The parameter `sec_prescaler` determines the value of the **secondary** SPI clock prescaler. This parameter is used only in the Master Mode. Valid settings are:
	
	`'--- secondary prescale (Master mode), SPI1CON1<4:2>`
	`const _SPI_PRESCALE_SEC_1 = $001C, ' secondary prescale 1:1`
	`const _SPI_PRESCALE_SEC_2 = $0018, ' secondary prescale 2:1`
	`const _SPI_PRESCALE_SEC_3 = $0014, ' secondary prescale 3:1`
	`const _SPI_PRESCALE_SEC_4 = $0010, ' secondary prescale 4:1`
	`const _SPI_PRESCALE_SEC_5 = $000C, ' secondary prescale 5:1`
	`const _SPI_PRESCALE_SEC_6 = $0008, ' secondary prescale 6:1`
	`const _SPI_PRESCALE_SEC_7 = $0004, ' secondary prescale 7:1`
	`const _SPI_PRESCALE_SEC_8 = $0000, ' secondary prescale 8:1`
	
	`' you can also use the following set of constants which are kept`
	`' for the sake of backward compatibility`
	
	`const _SPI_PRESCALE_1 = $001C ' value for prescale 1:1`
	`const _SPI_PRESCALE_2 = $0018 ' value for prescale 2:1`
	`const _SPI_PRESCALE_3 = $0014 ' value for prescale 3:1`
	`const _SPI_PRESCALE_4 = $0010 ' value for prescale 4:1`
	`const _SPI_PRESCALE_5 = $000C ' value for prescale 5:1`
	`const _SPI_PRESCALE_6 = $0008 ' value for prescale 6:1`
	`const _SPI_PRESCALE_7 = $0004 ' value for prescale 7:1`
	`const _SPI_PRESCALE_8 = $0000 ' value for prescale 8:1`
	`' continues ...`

Description (continued)

```
' continued ...
```

The parameter `pri_prescaler` determines the value of the **primary** SPI clock prescaler. This parameter is used only in the Master Mode. Valid settings are:

```
'--- primary prescale (Master mode), SPI1CON1<1:0>
const _SPI_PRESCALE_PRI_1        = $0003,   ' primary prescale 1:1
const _SPI_PRESCALE_PRI_4        = $0002,   ' primary prescale 4:1
const _SPI_PRESCALE_PRI_16       = $0001,   ' primary prescale 16:1
const _SPI_PRESCALE_PRI_64       = $0000,   ' primary prescale 64:1


' you can also use the following set of constants which are kept
' for the sake of backward compatibility
const _SPI_FCY_DIV1     = $0003    ' value for Fcy_div 1:1
const _SPI_FCY_DIV4     = $0002    ' value for Fcy_div 4:1
const _SPI_FCY_DIV16    = $0001    ' value for Fcy_div 16:1
const _SPI_FCY_DIV64    = $0000    ' value for Fcy_div 64:1
```

The parameter `slave_select` determines whether the Slave Select (SS) pin is used in communication. This parameter is valid in the Slave Mode only. It can be:

```
'--- Slave select enable bit (Slave Mode), SPI1CON1<7>
const _SPI_SS_ENABLE  = $0080,   ' SS used for the Slave mode
const _SPI_SS_DISABLE = $0000,   ' SS not used for the Slave mode
```

The parameter `data_sample` determines the sample moment (phase) of input data. It can be:

```
'--- SPI data input sample phase, SPI1CON1<9>
const _SPI_DATA_SAMPLE_MIDDLE    = $0000,
' data sampled in the middle of data output time
const _SPI_DATA_SAMPLE_END       = $2000,
' data sampled at end of data output time
```

The parameter `clock_idle` determines the behaviour of the SPI clock (CLK) line in IDLE phase:

```
'--- Clock Polarity Select bit, SPI1CON1<6>
const _SPI_CLK_IDLE_LOW          = $0000,   ' IDLE state is Lo,
                                            ' ACTIVE state is Hi
const _SPI_CLK_IDLE_HIGH         = $0040,   ' IDLE state is Hi,
                                             ' ACTIVE state is Lo

' continues...
```

Description (continued)	' continued.. The parameter edge determines on which clock edge the data is considered to be valid. Possible settings are: ' Clock EDGE select bit (where output data is valid), SPI1CON1<8> **const** _SPI_ACTIVE_2_IDLE = $0000, ' data is valid on ' ACTIVE-to-IDLE transition ' (data changes on IDLE-to-ACTIVE transition) **const** _SPI_IDLE_2_ACTIVE = $0100; ' data is valid on IDLE-to-ACTIVE transition ' (data changes on ACTIVE-to-IDLE transition)
Requires	MCU must have the SPI1 module.
Example	' Set SPI1 to the Master Mode, data length is 16-bit, clock = Fcy ' (no clock scaling), data sampled in the middle of interval, ' clock IDLE state high and data transmitted at low to high clock ' edge: Spi1_Init_Advanced(_SPI_MASTER, _SPI_16_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_1,_SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH,_SPI_ACTIVE_2_IDLE)

Spi1_Read

Prototype	**sub function** Spi1_Read(**dim** data_out **as word**) **as word**
Returns	Received data.
Description	Reads one word or byte (depending on mode set by init routines) from the SPI bus. Parameters : - data_out: dummy data for clock generation (see device Datasheet for SPI module implementation details)
Requires	MCU must have the SPI1 module. SPI1 must be initialized before using this function. See the Spi1_Init_Advanced and Spi1_Init routines.
Example	' read a byte from the SPI bus **dim** take, buffer **as byte** ... take = Spi1_Read(buffer)

Spi1_Write

Prototype	`sub procedure Spi1_Write(dim data_out as word)`
Description	Writes one word or byte (depending on mode set by init routines) via the SPI bus. Parameters : - `data_out`: data to be sent
Requires	MCU must have the SPI1 module. SPI1 must be initialized before using this function. See the Spi1_Init_Advanced and Spi1_Init routines.

Example	`' write a byte to the SPI bus`
	`dim buffer as byte`
	`...`
	`Spi1_Write(buffer)`

Spi2_Init

Prototype	`sub procedure Spi2_Init()`
Description	Configures and initializes the SPI2 module with default settings. Default settings: - Master mode - 8-bit data mode - Serial output data changes on transition from active clock state to Idle clock state secondary prescaler 1:1 - primary prescaler 64:1 - Slave Select disabled - input data sampled in the middle of interval - clock idle state low - Serial output data changes on transition from active clock state to idle clock state
Requires	MCU must have the SPI2 module.
Example	`' Initialize the SPI2 module with default settings` `Spi2_Init()`

Spi2_Init_Advanced

Prototype	`sub procedure Spi2_Init_Advanced(dim master_mode, mode16, sec_prescaler, pri_prescaler, slave_select, data_sample, clock_idle, edge as word)`

Description	Configures and initializes the SPI2 module.
	
	The `master_mode` parameter determines the working mode for SPI2. It can have the following values:
	
	`'--- Master/Slave mode, SPI2CON1<5>`
	`const _SPI_MASTER = $0020, ' Master mode`
	`const _SPI_SLAVE = $0000, ' Slave mode`
	
	The parameter `mode16` determines the data length mode, which can be 8-bits (per transmitions cycle) or 16-bits. It is being configured as follows:
	
	`'--- data length select, SPI2CON1<10>`
	`const _SPI_16_BIT = $0400, ' 16-bit mode`
	`const _SPI_8_BIT = $0000, ' 8-bit mode`
	
	The parameter `sec_prescaler` determines the value of the **secondary** SPI clock prescaler. This parameter is used only in the Master Mode. Valid settings are:
	
	`'--- secondary prescale (Master mode), SPI2CON1<4:2>`
	`const _SPI_PRESCALE_SEC_1 = $001C, ' secondary prescale 1:1`
	`const _SPI_PRESCALE_SEC_2 = $0018, ' secondary prescale 2:1`
	`const _SPI_PRESCALE_SEC_3 = $0014, ' secondary prescale 3:1`
	`const _SPI_PRESCALE_SEC_4 = $0010, ' secondary prescale 4:1`
	`const _SPI_PRESCALE_SEC_5 = $000C, ' secondary prescale 5:1`
	`const _SPI_PRESCALE_SEC_6 = $0008, ' secondary prescale 6:1`
	`const _SPI_PRESCALE_SEC_7 = $0004, ' secondary prescale 7:1`
	`const _SPI_PRESCALE_SEC_8 = $0000, ' secondary prescale 8:1`
	
	`' you can also use the following set of constants which are kept`
	`' for the sake of backward compatibility`
	`const _SPI_PRESCALE_1 = $001C ' value for prescale 1:1`
	`const _SPI_PRESCALE_2 = $0018 ' value for prescale 2:1`
	`const _SPI_PRESCALE_3 = $0014 ' value for prescale 3:1`
	`const _SPI_PRESCALE_4 = $0010 ' value for prescale 4:1`
	`const _SPI_PRESCALE_5 = $000C ' value for prescale 5:1`
	`const _SPI_PRESCALE_6 = $0008 ' value for prescale 6:1`
	`const _SPI_PRESCALE_7 = $0004 ' value for prescale 7:1`
	`const _SPI_PRESCALE_8 = $0000 ' value for prescale 8:1`
	
	`' continues ...`

Description (continued)

```
' continued ...
```

The parameter `pri_prescaler` determines the value of the **primary** SPI clock prescaler. This parameter is used only in the Master Mode. Valid settings are:

```
'--- primary prescale (Master mode), SPI2CON1<1:0>
const _SPI_PRESCALE_PRI_1        = $0003,   ' primary prescale 1:1
const _SPI_PRESCALE_PRI_4        = $0002,   ' primary prescale 4:1
const _SPI_PRESCALE_PRI_16       = $0001,   ' primary prescale 16:1
const _SPI_PRESCALE_PRI_64       = $0000,   ' primary prescale 64:1

' you can also use the following set of constants which are kept
' for the sake of backward compatibility

const _SPI_FCY_DIV1      = $0003;   ' value for Fcy_div 1:1
const _SPI_FCY_DIV4      = $0002;   ' value for Fcy_div 4:1
const _SPI_FCY_DIV1      = $0001;   ' value for Fcy_div 16:1
const _SPI_FCY_DIV64     = $0000;   ' value for Fcy_div 64:1
```

The parameter `slave_select` determines whether the Slave Select (SS) pin is used in communication. This parameter is valid in the Slave Mode only. It can be:

```
'--- Slave select enable bit (Slave Mode), SPI2CON1<7>
const _SPI_SS_ENABLE    = $0080,   ' SS used for the Slave mode
const _SPI_SS_DISABLE   = $0000,   ' SS not used for the Slave mode
```

The parameter `data_sample` determines the sample moment (phase) of input data. It can be:

```
'--- SPI data input sample phase, SPI2CON1<9>
const _SPI_DATA_SAMPLE_MIDDLE    = $0000,
' data sampled in the middle of data output time

const _SPI_DATA_SAMPLE_END       = $2000,
' data sampled at end of data output time
```

The parameter `clock_idle` determines the behaviour of the SPI clock (CLK) line in IDLE phase:

```
'--- Clock Polarity Select bit, SPI2CON1<6>
const _SPI_CLK_IDLE_LOW          = $0000,
' IDLE state is Lo, ACTIVE state is Hi

const _SPI_CLK_IDLE_HIGH         = $0040,
' IDLE state is Hi, ACTIVE state is Lo

' continues...
```

Description (continued)	' continued.. The parameter `edge` determines on which clock edge data is considered to be valid. Possible settings are: ' Clock EDGE select bit (where output data is valid), SPI2CON1<8> **const** _SPI_ACTIVE_2_IDLE = $0000, ' data is valid on ACTIVE-to-IDLE transition ' (data changes on IDLE-to-ACTIVE transition) **const** _SPI_IDLE_2_ACTIVE = $0100; ' data is valid on IDLE-to-ACTIVE transition ' (data changes on ACTIVE-to-IDLE transition)
Requires	MCU must have the SPI2 module.
Example	`Spi2_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_64,_SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_2_IDLE)`

Spi2_Read

Prototype	**sub function** Spi2_Read(**dim** data_out **as word**) **as word**
Returns	Returns the received data.
Description	Reads one word or byte (depending on mode set by init routines) from the SPI bus. Parameters : - `data_out`: dummy data for clock generation (see device Datasheet for SPI module implementation details).
Requires	MCU must have the SPI2 module. SPI2 must be initialized before using this function. See the Spi2_Init_Advanced and Spi2_Init routines.
Example	' read a byte from the SPI bus **dim** take, buffer **as byte** ... take = Spi2_Read(buffer)

Spi2_Write

Prototype	`sub procedure Spi2_Write(dim data_out as word)`
Description	Writes one word or byte (depending on mode set by init routines) via the SPI bus. Parameters : - `data_out`: data to be sent
Requires	MCU must have the SPI2 module. SPI2 must be initialized before using this function. See the Spi2_Init_Advanced and Spi2_Init routines.

Example	`' write a byte to the SPI bus`
	`dim buffer as byte`
	`...`
	`Spi2_Write(buffer)`

Spi_Init

Prototype	`sub procedure Spi_Init()`
Description	Configures and initializes the SPI1 module with default settings. Default settings: - Master mode - 8-bit data mode - secondary prescaler 1:1 - primary prescaler 64:1 - Slave Select disabled - input data sampled in the middle of interval - clock idle state low - Serial output data changes on transition from active clock state to idle clock state
Requires	MCU must have the SPI1 module.
Example	`' Initialize the SPI1 module with default settings` `Spi_Init()`

Spi_Init_Advanced

Prototype	`sub procedure Spi_Init_Advanced(dim master_mode, mode16, sec_prescaler, pri_prescaler, slave_select, data_sample, clock_idle, edge as word)`

Description	Configures and initializes the SPI1 module.
	
	The `master_mode` parameter determines the working mode for SPI1. It can have the following values:
	
	`'--- Master/Slave mode, SPI1CON1<5>`
	`const _SPI_MASTER = $0020, ' Master mode`
	`const _SPI_SLAVE = $0000, ' Slave mode`
	
	The parameter `mode16` determines the data length mode, which can be 8-bits (per transmitions cycle) or 16-bits. It is being configured as follows:
	
	`'--- data length select, SPI1CON1<10>`
	`const _SPI_16_BIT = $0400, ' 16-bit mode`
	`const _SPI_8_BIT = $0000, ' 8-bit mode`
	
	The parameter `sec_prescaler` determines the value of the **secondary** SPI clock prescaler. This parameter is used only in the Master Mode. Valid settings are:
	`'--- secondary prescale (Master mode), SPI1CON1<4:2>`
	`const _SPI_PRESCALE_SEC_1 = $001C, ' secondary prescale 1:1`
	`const _SPI_PRESCALE_SEC_2 = $0018, ' secondary prescale 2:1`
	`const _SPI_PRESCALE_SEC_3 = $0014, ' secondary prescale 3:1`
	`const _SPI_PRESCALE_SEC_4 = $0010, ' secondary prescale 4:1`
	`const _SPI_PRESCALE_SEC_5 = $000C, ' secondary prescale 5:1`
	`const _SPI_PRESCALE_SEC_6 = $0008, ' secondary prescale 6:1`
	`const _SPI_PRESCALE_SEC_7 = $0004, ' secondary prescale 7:1`
	`const _SPI_PRESCALE_SEC_8 = $0000, ' secondary prescale 8:1`
	
	`' you can also use the following set of constants which are kept`
	`' for the sake of backward compatibility`
	
	`const _SPI_PRESCALE_1 = $001C ' value for prescale 1:1`
	`const _SPI_PRESCALE_2 = $0018 ' value for prescale 2:1`
	`const _SPI_PRESCALE_3 = $0014 ' value for prescale 3:1`
	`const _SPI_PRESCALE_4 = $0010 ' value for prescale 4:1`
	`const _SPI_PRESCALE_5 = $000C ' value for prescale 5:1`
	`const _SPI_PRESCALE_6 = $0008 ' value for prescale 6:1`
	`const _SPI_PRESCALE_7 = $0004 ' value for prescale 7:1`
	`const _SPI_PRESCALE_8 = $0000 ' value for prescale 8:1`
	
	`' continues ...`

Description (continued)

```
' continued ...
```

The parameter `pri_prescaler` determines the value of the **primary** SPI clock prescaler. This parameter is used only in the Master Mode. Valid settings are:

```
'--- primary prescale (Master mode), SPI1CON1<1:0>
const _SPI_PRESCALE_PRI_1        = $0003,   ' primary prescale 1:1
const _SPI_PRESCALE_PRI_4        = $0002,   ' primary prescale 4:1
const _SPI_PRESCALE_PRI_16       = $0001,   ' primary prescale 16:1
const _SPI_PRESCALE_PRI_64       = $0000,   ' primary prescale 64:1

' you can also use the following set of constants which are kept
' for the sake of backward compatibility

const _SPI_FCY_DIV1      = $0003;   ' value for Fcy_div 1:1
const _SPI_FCY_DIV4      = $0002;   ' value for Fcy_div 4:1
const _SPI_FCY_DIV1      = $0001;   ' value for Fcy_div 16:1
const _SPI_FCY_DIV64     = $0000;   ' value for Fcy_div 64:1
```

The parameter `slave_select` determines whether the Slave Select (SS) pin is used in communication. This parameter is valid in the Slave Mode only. It can be:

```
'--- Slave select enable bit (Slave Mode), SPI1CON1<7>
const _SPI_SS_ENABLE    = $0080,   ' SS used for the Slave mode
const _SPI_SS_DISABLE   = $0000,   ' SS not used for the Slave mode
```

The parameter `data_sample` determines the sample moment (phase) of input data. It can be:

```
'--- SPI data input sample phase, SPI1CON1<9>
const _SPI_DATA_SAMPLE_MIDDLE    = $0000,
' data sampled in the middle of data output time

const _SPI_DATA_SAMPLE_END       = $2000,
' data sampled at end of data output time
```

The parameter `clock_idle` determines the behaviour of the SPI clock (CLK) line in IDLE phase:

```
'--- Clock Polarity Select bit, SPI1CON1<6>
const _SPI_CLK_IDLE_LOW          = $0000,
' IDLE state is Lo, ACTIVE state is Hi

const _SPI_CLK_IDLE_HIGH         = $0040,
' IDLE state is Hi, ACTIVE state is Lo

' continues...
```

Description (continued)	' continued.. The parameter `edge` determines on which clock edge data is considered to be valid. Possible settings are: ' Clock EDGE select bit (where output data is valid), SPI1CON1<8> **const** _SPI_ACTIVE_2_IDLE = $0000, ' data is valid on ACTIVE-to-IDLE transition ' (data changes on IDLE-to-ACTIVE transition) **const** _SPI_IDLE_2_ACTIVE = $0100; ' data is valid on IDLE-to-ACTIVE transition ' (data changes on ACTIVE-to-IDLE transition)
Requires	MCU must have the SPI1 module.
Example	' Set SPI1 to the Master Mode, data length is 16-bit, clock = Fcy ' (no clock scaling), data sampled in the middle of interval, ' clock IDLE state high and data transmitted at low to high clock ' edge: Spi_Init_Advanced(_SPI_MASTER, _SPI_16_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_1,_SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH,_SPI_ACTIVE_2_IDLE)

Spi_Read

Prototype	**sub function** Spi_Read(**dim** data_out **as word**) **as word**
Returns	Received data.
Description	Reads one word or byte (depending on mode set by init routines) from the SPI bus. Parameters : - `data_out`: dummy data for clock generation (see device Datasheet for SPI modules implementation details) **Note**: if MCU has 2 SPI modules, active module will be used. See the Spi_Set_Active routine.
Requires	Routine requires at least one SPI module. Used SPI module must be initialized before using this function. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.
Example	' read a byte from the SPI bus **dim** take, buffer **as byte** ... take = Spi_Read(buffer)

Spi_Write

Prototype	`sub procedure Spi_Write(dim data_out as word)`
Description	Writes one word or byte (depending on mode set by init routines) via the SPI bus. Parameters : - `data_out`: data to be sent **Note**: if MCU has 2 SPI modules, active module will be used. See the Spi_Set_Active routine.
Requires	Routine requires at least one SPI module. Used SPI module must be initialized before using this function. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.

Example	`' write a byte to the SPI bus`
	`dim buffer as byte`
	`...`
	`Spi_Write(buffer)`

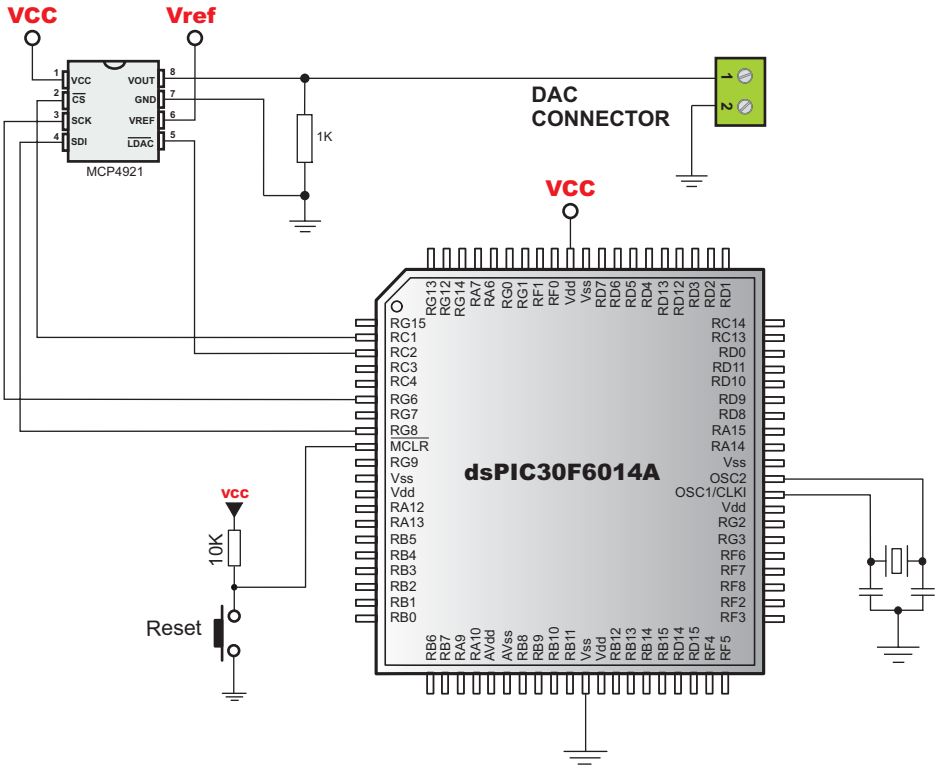
Spi_Set_Active

Prototype	`sub procedure Spi_Set_Active(dim SpiNo as byte))`
Description	Sets the active SPI module which will be used by the Spi_Read and Spi_Write routines. Parameters : - `SpiNo`: module number. Valid values: 1 (for SPI1) and 2 (for SPI2)
Requires	Routine is available only for MCUs with two SPI modules. Used SPI module must be initialized before using this function. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.
Example	`Spi_Set_Active(2)  ' Sets the SPI2 module active`

Library Example

The code demonstrates how to use SPI library functions for communication between the SPI2 module of the MCU and MCP4921 DAC chip

```
program SPITest
const  LOAD_PIN       = 2   ' DAC load pin
const  CS_PIN         = 1   ' DAC CS pin

dim value as word

sub procedure InitMain()
  TRISC.LOAD_PIN = 0                      ' LOAD pin
  TRISC.CS_PIN   = 0                      ' CS pin
  LATC.CS_PIN    = 1                      ' Set CS to inactive
  LATC.LOAD_PIN  = 0                      ' Set LOAD to inactive
  Spi2_Init_Advanced(_SPI_MASTER, _SPI_16_BIT, _SPI_PRESCALE_SEC_1,
_SPI_PRESCALE_PRI_1, _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE,
_SPI_CLK_IDLE_HIGH,_SPI_ACTIVE_2_IDLE)    ' Init SPI
end sub

' DAC increments (0..4095) --> output voltage (0..Vref)
sub procedure DAC_Output(dim valueDAC as word)
  while (SPI2STAT.1 = 1)     ' wait for SPI module to finish, if doing something
    nop
  wend
  LATC.CS_PIN = 0                   ' CS enable for DAC
  SPI2BUF = 0x3000 or valueDAC
  ' Write CurrentValue to DAC (0x3 is required by DAC)
  while (SPI2STAT.1 = 1)           ' Wait for SPI module to finish write
    nop
  wend

  LATC.CS_PIN    = 1               ' CS disable for DAC
end sub

main:
  InitMain()
  while true
    value = 1
    while value < $FFF
      DAC_Output(value)
      Delay_ms(5)
      value = value+1
    wend
  wend                              '   the output in the mid-range
end.
```


VCC
Vref
VCC
CS
SCK
SDI
VOUT
GND
VREF
LDAC
MCP4921
1K
DAC
CONNECTOR
VCC
dsPIC30F6014A
RG15
RC1
RC2
RC3
RC4
RG6
RG7
RG8
MCLR
RG9
Vss
Vdd
RA12
RA13
RB5
RB4
RB3
RB2
RB1
RB0
RC14
RC13
RD0
RD11
RD10
RD9
RD8
RA15
RA14
Vss
OSC2
OSC1/CLKI
Vdd
RG2
RG3
RF6
RF7
RF8
RF2
RF3
VCC
10K
Reset

SPI Ethernet Library

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

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

This library is designed to simplify handling of the underlying hardware (ENC28J60). It works with any dsPIC30/33 and PIC24 with integrated SPI and more than 4 Kb ROM. 32 to 40 MHz clock is recommended to get from 8 to 10 MHz SPI clock, otherwise the dsPIC30/33 and PIC24 should be clocked by ENC clock output due to ENC silicone bug in the SPI hardware. If you try lower dsPIC30/33 and PIC24 clock speed, there might be a board hang or some requests might be missed.

Note: For advanced users there is a header file (`enc28j60_libprivate.dbas`) in the Uses folder of the compiler with detailed description of all functions which are implemented in the SPI Ethernet Library.

Note: The Library uses the SPI module for communication. The user must initialize the appropriate SPI module before using the SPI Ethernet Library. For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `Spi_Set_Active()` function. See the Spi Library functions.

Library Routines

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

SPI_Ethernet_Init

Prototype	`sub procedure SPI_Ethernet_Init(dim byref resetPort as word, dim resetBit as word, dim byref CSportPtr as word, dim CSbit as word, dim byref mac as byte[6], dim byref ip as byte[4], dim fullDuplex as byte)`
Description	This function initializes ENC controller. This function is split into 2 parts in order to help linker when coming short of memory. Parameters: - `resetPort`: reset signal port address - `resetBit`: reset signal pin - `CSportPtr`: chip select signal port address - `CSbit`: chip select signal pin - `mac`: 6 char array containing MAC address - `ip`: 4 char array containing IP address - `fullDuplex`: communication mode parameter. Valid values: `SPI_Ethernet_HALFDUPLEX` and `SPI_Ethernet_FULLDUPLEX`
Requires	The SPI module needs to be initialized. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.
Example	`Spi1_Init()` `SPI_Ethernet_Init(PORTF, 0, PORTF, 1, myMacAddr, myIpAddr, SPI_Ethernet_FULLDUPLEX)`

SPI_Ethernet_doPacket

Prototype	`sub procedure SPI_Ethernet_doPacket()`
Description	The function checks if a packet has been received. If it has, the function processes it in the following manner: - reply to ARP & ICMP echo requests automatically - upon TCP request the SPI_Ethernet_UserTCP function is called for further processing - upon UDP request the SPI_Ethernet_UserUDP function is called for further processing **Note**: SPI_Ethernet_doPacket must be called as often as possible in user code.
Requires	Ethernet module has to be initialized . See SPI_Ethernet_Init.
Example	`SPI_Ethernet_doPacket()`

SPI_Ethernet_putByte

Prototype	`sub procedure SPI_Ethernet_putByte(dim v as byte)`
Description	Stores one byte to address pointed by the ENC28J60 write pointer (`EWRPT`). Parameters: - `v`: value to store
Requires	Ethernet module has to be initialized . See SPI_Ethernet_Init.

Example	`dim data as byte`
	`...`
	`SPI_Ethernet_putByte(data)`

SPI_Ethernet_getByte

Prototype	`sub function SPI_Ethernet_getByte() as byte`
Returns	Byte read from ENC28J60 memory.
Description	Gets byte from address pointed by the ENC28J60 read pointer (`ERDPT`).
Requires	Ethernet module has to be initialized . See SPI_Ethernet_Init.
Example	`dim buffer as byte` `buffer = SPI_Ethernet_getByte()`

SPI_Ethernet_UserTCP

Prototype	`sub function SPI_Ethernet_UserTCP(dim byref remoteHost as byte[4], dim remotePort, localPort, reqLength as word) as word`
Returns	- `0` - if there should not be a reply to the request. - Length of TCP/HTTP reply data field, otherwise
Description	This function is internally called by the library. The user accesses to the TCP/HTTP request by successive calls to SPI_Ethernet_getByte(). The user puts data in the transmit buffer by successive calls to SPI_Ethernet_putByte(). The function must return the length in bytes of the TCP/HTTP reply, or 0 if there is nothing to transmit. If there is no need to reply to the TCP/HTTP requests, just define this function with return(0) as a single statement. Parameters: - `remoteHost`: client's IP address - `remotePort`: client's TCP port - `localPort`: port to which the request is sent - `reqLength`: TCP/HTTP request data field length **Note**: The function source code is provided with appropriate example projects. The code should be adjusted by the user to achieve desired reply.
Requires	Ethernet module has to be initialized . See SPI_Ethernet_Init.
Example	This function is internally called by the library.

SPI_Ethernet_UserUDP

Prototype	`sub function SPI_Ethernet_UserUDP(dim byref remoteHost as byte[ 4] , dim remotePort, destPort, reqLength as word) as word`
Returns	Length of UDP reply data field, or 0 if the library should not reply to the request.
Description	This function is internally called by the library. The user accesses to the UDP request by successive calls to SPI_Ethernet_getByte(). The user puts data in the transmit buffer by successive calls to SPI_Ethernet_putByte(). The function must return the length in bytes of the UDP reply, or 0 if nothing to transmit. If you don't need to reply to the UDP requests, just define this function with a return(0) as single statement. Parameters: - `remoteHost`: client's IP address - `remotePort`: client's port - `destPort`: port to which the request is sent - `reqLength`: UDP request data field length **Note**: The function source code is provided with appropriate example projects. The code should be adjusted by the user to achieve desired reply.
Requires	Ethernet module has to be initialized . See SPI_Ethernet_Init.
Example	This function is internally called by the library.

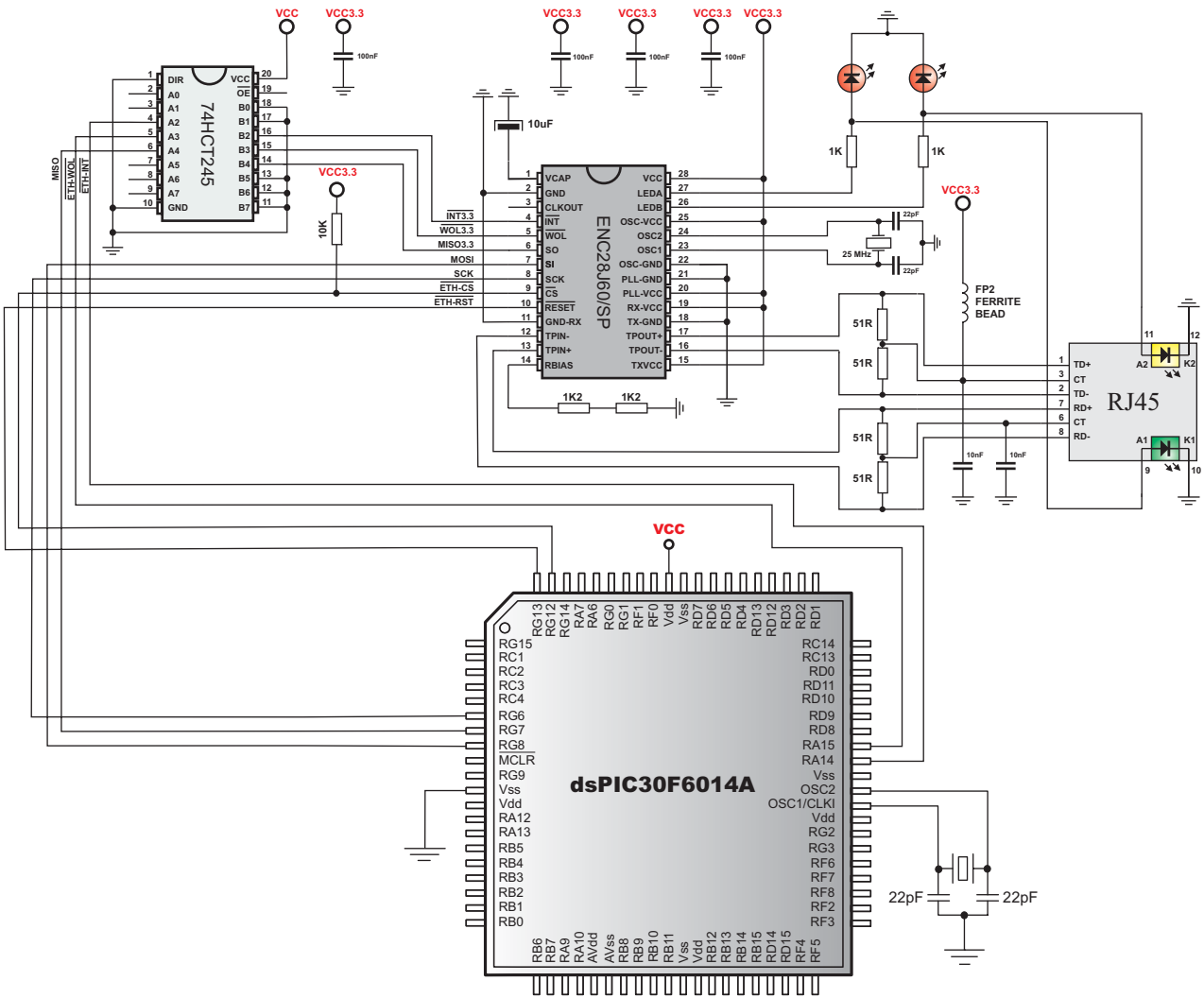
Library Example

The following example is a simple demonstration of the SPI Ethernet Library. IP address 192.168.20.60 is assigned to the MCU+ENC28J60 server. The server will respond to ping, the TCP and UDP requests if connected to a local area network. Hardware configurations in this example are made for the dsPICPRO2 board and dsPIC30F6014A.

```
program enc_ethernet

include enc_utils ' this is where you should write implementation for UDP and HTTP


  '************************************
  '* RAM variables
  '*
dim myMacAddr    as byte[ 6]       ' my MAC address
    myIpAddr     as byte[ 4]       ' my IP address

main:
  ADPCFG = 0xFBFF                  ' ADC convertors will be used (channel10)

  PORTB = 0
  TRISB = 0xFFFF                   ' set PORTB as input for buttons

  PORTD = 0x00
  TRISD = 0                        ' set PORTD as output

  httpCounter = 0

  myMacAddr[ 0]  = 0x00
  myMacAddr[ 1]  = 0x14
  myMacAddr[ 2]  = 0xA5
  myMacAddr[ 3]  = 0x76
  myMacAddr[ 4]  = 0x19
  myMacAddr[ 5]  = 0x3F

  myIpAddr[ 0]  = 192
  myIpAddr[ 1]  = 168
  myIpAddr[ 2]  = 20
  myIpAddr[ 3]  = 60



' continues...
```

```
  ' continued...


  '*
  '* starts ENC28J60 with :
  '* reset bit on RF0
  '* CS bit on RF1
  '* my MAC & IP address
  '* full duplex
  '*

  ' for faster spi communication use Spi_Init_Advanced routine
  Spi2_Init()
  Spi_Ethernet_Init(PORTG, 13, PORTG, 12, myMacAddr, myIpAddr, SPI_Ethernet_FULLDU-
PLEX)
  while true                    ' do forever
    Spi_Ethernet_doPacket()     ' process incoming Ethernet packets

    '*
    '* add your stuff here if needed
    '* ENC28J60_doPacket() must be called as often as possible
    '* otherwise packets could be lost
    '*
  wend
end.
```

Hardware Connection



SPI Graphic LCD Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for operating Graphic LCD 128x64 (with the commonly used Samsung KS108/KS107 controller) via SPI interface. For creating a custom set of GLCD images use the GLCD Bitmap Editor Tool.

Note: When using this library with the dsPIC33 and PIC24 family MCUs be aware of their voltage incompatibility with a certain number of Samsung KS0108 based GLCD modules. So, additional external power supply for these modules may be required.

Note: The library uses the SPI module for communication. The user must initialize the appropriate SPI module before using the SPI GLCD Library. For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `Spi_Set_Active()` function. See the Spi Library functions.

Note: This Library is designed to work with the mikroElektronika's Serial LCD/GLCD Adapter Board pinout, see schematic at the bottom of this page for details.

Library Routines

Basic routines:

```
SPI_Glcd_Config
Spi_Glcd_Init
Spi_Glcd_Set_Side
Spi_Glcd_Set_Page
Spi_Glcd_Set_X
Spi_Glcd_Read_Data
Spi_Glcd_Write_Data
```

Advanced routines:

```
Spi_Glcd_Fill
Spi_Glcd_Dot
Spi_Glcd_Line
Spi_Glcd_V_Line
Spi_Glcd_H_Line
Spi_Glcd_Rectangle
Spi_Glcd_Box
Spi_Glcd_Circle
Spi_Glcd_Set_Font
Spi_Glcd_Write_Char
Spi_Glcd_Write_Text
Spi_Glcd_Image
```

Spi_Glcd_Config

Prototype	`sub procedure Spi_Glcd_Config(dim DeviceAddress as byte, dim byref RstPort as word, dim RstPin as word, dim byref CSPort as word, dim CSPin as word)`
Description	Initializes the GLCD module via SPI interface. Parameters : - `DeviceAddress`: spi expander hardware address, see schematic at the bottom of this page - `RstPort`: reset signal port address - `RstPin`: reset signal pin - `CSPort`: chip select signal port address - `CSPin`: chip select signal pin
Requires	The SPI module needs to be initialized. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.

Example	`' initialize the SPI1 module and GLCD module`
	`Spi_Init()`
	`Spi_Glcd_Config(0, PORTF, 0, PORTF, 1)`

Spi_Glcd_Config

Prototype	`sub procedure Spi_Glcd_Init()`
Description	Initializes the GLCD module via SPI interface with the `DeviceAddress=0`, `reset` signal connected to the `RF0` pin and the `chip select` signal connected to the `RF1` pin. **Note**: The `Spi_Glcd_Init()` routine can be changed by the user. Source code is located in the setup library file in the Uses folder.
Requires	The SPI module needs to be initialized. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.

Example	`' initialize the SPI1 module and GLCD module`
	`Spi_Init()`
	`Spi_Glcd_Init()`

Spi_Glcd_Set_Side

Prototype	`sub procedure Spi_Glcd_Set_Side(dim x_pos as byte)`
Description	Selects GLCD side. Refer to the GLCD datasheet for detail explanation. Parameters : - `x_pos`: position on x-axis. Valid values: 0..127 The parameter x_pos specifies the GLCD side: values from 0 to 63 specify the left side, values from 64 to 127 specify the right side. **Note**: For side, x axis and page layout explanation, see schematic at the bottom of this page.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.

Example	The following two lines are equivalent, and both of them select the left side of GLCD:
	
	`SPI_Glcd_Set_Side(0)`
	`SPI_Glcd_Set_Side(10)`

Spi_Glcd_Set_Page

Prototype	`sub procedure Spi_Glcd_Set_Page(dim page as byte)`
Description	Selects page of GLCD. Parameters : - `page`: page number. Valid values: 0..7 **Note**: For side, x axis and page layout explanation, see schematic at the bottom of this page.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`Spi_Glcd_Set_Page(5)`

Spi_Glcd_Set_X

Prototype	`sub procedure Spi_Glcd_Set_X(dim x_pos as byte)`
Description	Sets x-axis position to x_pos dots from the left border of GLCD within the selected side. Parameters : - `x_pos`: position on x-axis. Valid values: 0..63 **Note**: For side, x axis and page layout explanation, see schematic at the bottom of this page.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`Spi_Glcd_Set_X(25)`

Spi_Glcd_Read_Data

Prototype	`sub function Spi_Glcd_Read_Data() as byte`
Returns	One byte from GLCD memory.
Description	Reads data from the current location of GLCD memory and moves to the next location.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines. GLCD side, x-axis position and page should be set first. See the functions Spi_Glcd_Set_Side, Spi_Glcd_Set_X, and Spi_Glcd_Set_Page.

Example	`dim data as byte`
	`...`
	`data = Spi_Glcd_Read_Data()`

Spi_Glcd_Write_Data

Prototype	`sub procedure Spi_Glcd_Write_Data(dim data as byte)`
Description	Writes one byte to the current location in GLCD memory and moves to the next location. Parameters : - `data`: data to be written.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines. GLCD side, x-axis position and page should be set first. See the functions Spi_Glcd_Set_Side, Spi_Glcd_Set_X, and Spi_Glcd_Set_Page.

Example	`dim data as byte`
	`...`
	`Spi_Glcd_Write_Data(data)`

Spi_Glcd_Fill

Prototype	`sub procedure Spi_Glcd_Fill(dim pattern as byte)`
Description	Fills GLCD memory with byte pattern. Parameters : - `pattern`: byte to fill GLCD memory with To clear the GLCD screen, use `Spi_Glcd_FillGlcd_Fill(0)`. To fill the screen completely, use `Spi_Glcd_Fill(0xFF)`.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`' Clear screen` `Spi_Glcd_Fill(0)`

Spi_Glcd_Dot

Prototype	`sub procedure Spi_Glcd_Dot(dim x_pos, y_pos, color as byte)`
Description	Draws a dot on GLCD at coordinates (x_pos, y_pos). Parameters : - `x_pos`: x position. Valid values: 0..127 - `y_pos`: y position. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state. **Note**: For x and y axis layout explanation, see schematic at the bottom of this page.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`' Invert the dot in the upper left corner` `Spi_Glcd_Dot(0, 0, 2)`

Spi_Glcd_Line

Prototype	`sub procedure SPI_Glcd_Line(dim x_start, y_start, x_end, y_end as integer, dim color as byte)`
Description	Draws a line on GLCD. Parameters : - `x_start`: x coordinate of the line start. Valid values: 0..127 - `y_start`: y coordinate of the line start. Valid values: 0..63 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 Parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`' Draw a line between dots (0,0) and (20,30)` `Spi_Glcd_Line(0, 0, 20, 30, 1)`

Spi_Glcd_V_Line

Prototype	`sub procedure Spi_Glcd_V_Line(dim y_start, y_end, x_pos, color as byte)`
Description	Draws a vertical line on GLCD. Parameters : - `y_start`: y coordinate of the line start. Valid values: 0..63 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `x_pos`: x coordinate of vertical line. Valid values: 0..127 - `color`: color parameter. Valid values: 0..2 The parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`' Draw a vertical line between dots (10,5) and (10,25)` `Spi_Glcd_V_Line(5, 25, 10, 1)`

Spi_Glcd_H_Line

Prototype	`sub procedure Spi_Glcd_H_Line(dim x_start, x_end, y_pos, color as byte)`
Description	Draws a horizontal line on GLCD. Parameters : - `x_start`: x coordinate of the line start. Valid values: 0..127 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_pos`: y coordinate of horizontal line. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`' Draw a horizontal line between dots (10,20) and (50,20)` `Spi_Glcd_H_Line(10, 50, 20, 1)`

Spi_Glcd_Rectangle

Prototype	`sub procedure SPI_Glcd_Rectangle(dim x_upper_left, y_upper_left, x_bottom_right, y_bottom_right, color as byte)`
Description	Draws a rectangle on GLCD. Parameters : - `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`:x coordinate of the lower right rectangle corner.Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`' Draw a rectangle between dots (5,5) and (40,40)` `Spi_Glcd_Rectangle(5, 5, 40, 40, 1)`

Spi_Glcd_Box

Prototype	`sub procedure Spi_Glcd_Box(dim x1, y1, x2, y2, color as byte)`
Description	Draws a box on GLCD. Parameters : - `x_upper_left`: x coordinate of the upper left box corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left box corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right box corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right box corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the box fill: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`' Draw a box between dots (5,15) and (20,40)` `Spi_Glcd_Box(5, 15, 20, 40, 1)`

Spi_Glcd_Circle

Prototype	`sub procedure SPI_Glcd_Circle(dim x_center, y_center, radius as integer, dim color as byte)`
Description	Draws a circle on GLCD. Parameters : - `x_center`: x coordinate of the circle center. Valid values: 0..127 - `y_center`: y coordinate of the circle center. Valid values: 0..63 - `radius`: radius size - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the circle line: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`' Draw a circle with center in (50,50) and radius=10` `Spi_Glcd_Circle(50, 50, 10, 1)`

Spi_Glcd_Set_Font

Prototype	`sub procedure SPI_Glcd_Set_Font(dim activeFont as LongInt, dim aFontWidth, aFontHeight as byte, dim aFontOffs as word)`
Description	Sets font that will be used with the Spi_Glcd_Write_Char and Spi_Glcd_Write_Text routines. Parameters : - `activeFont`: font address. Font needs to be formatted as an array of bytes - `aFontWidth`: width of the font characters in dots. - `aFontHeight`: height of the font characters in dots. - `aFontOffs`: number that represents difference between the mikroBasic character set and regular ASCII set (eg. if 'A' is 65 in ASCII character, and 'A' is 45 in the mikroBasic character set, aFontOffs is 20). Demo fonts supplied with the library have an offset of 32, which means that they start with space. The user can use fonts given in the file "__Lib_Glcd_Fonts.dbas" file located in the Uses\Consts folder or create his/her own fonts.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`' Use the custom 5x7 font "myfont" which starts with space (32):` `Spi_Glcd_Set_Font(@myfont, 5, 7, 32)`

Spi_Glcd_Write_Char

Prototype	`sub procedure SPI_Glcd_Write_Char(dim chr1, x_pos, page_num, color as byte)`
Description	Prints character on GLCD. Parameters : - `chr1`: character to be written - `x_pos`: character starting position on x-axis. Valid values: 0..(127-FontWidth) - `page_num`: the number of the page on which character will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the character: 0 white, 1 black, and 2 inverts each dot. **Note**: For x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines. Use Spi_Glcd_Set_Font to specify the font for display; if no font is specified, then the default 5x8 font supplied with the library will be used.
Example	`' Write character 'C' on the position 10 inside the page 2:` `Spi_Glcd_Write_Char('C', 10, 2, 1)`

Spi_Glcd_Write_Text

Prototype	`sub procedure SPI_Glcd_Write_Text(dim byref text as char[40], dim x_pos, page_num, color as byte)`
Description	Prints text on GLCD. Parameters : - `text`: text to be written - `x_pos`: text starting position on x-axis. - `page_num`: the number of the page on which text will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the text: 0 white, 1 black, and 2 inverts each dot. **Note**: For x axis and page layout explanation, see schematic at the bottom of this page.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines. Use Spi_Glcd_Set_Font to specify the font for display; if no font is specified, then the default 5x8 font supplied with the library will be used.
Example	`' Write text "Hello world!" on the position 10 inside the page 2:` `Spi_Glcd_Write_Text("Hello world!", 10, 2, 1)`

Spi_Glcd_Image

Prototype	`sub procedure SPI_Glcd_Image(dim byref constimage as byte)`
Description	Displays bitmap on GLCD. Parameters : - `image`: image to be displayed. Bitmap array can be located in both code and RAM (due to the mikroBasic for dsPIC30/33 and PIC24 pointer to const and pointer to RAM equivalency). Use the mikroBasic's integrated GLCD Bitmap Editor (menu option **Tools › GLCD Bitmap Editor**) to convert image to a constant array suitable for displaying on GLCD.
Requires	GLCD needs to be initialized for SPI communication, see the SPI_Glcd_Config and Spi_Glcd_Init routines.
Example	`Spi_Glcd_Image(my_image)`

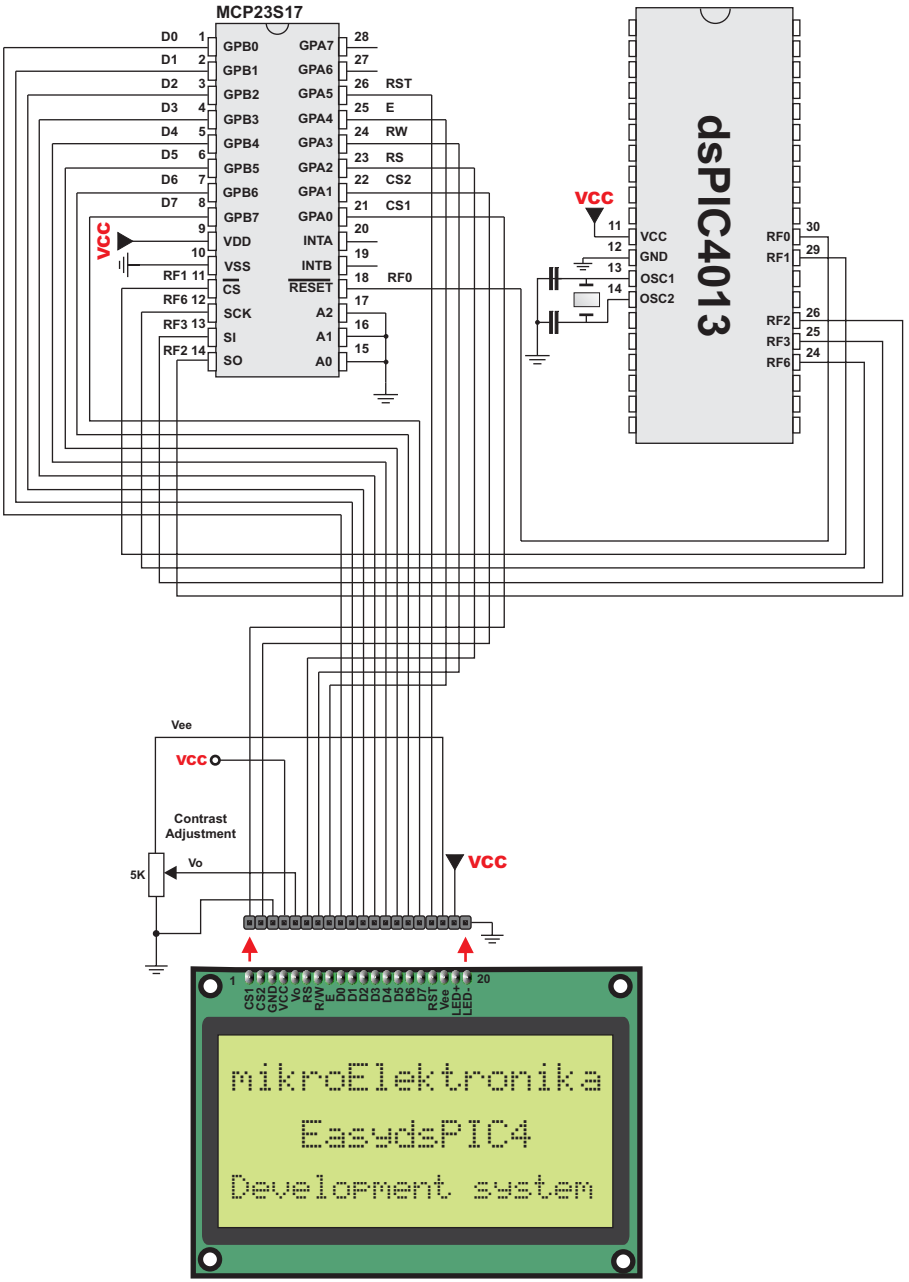
Library Example

The example demonstrates how to communicate to KS0108 GLCD via SPI module, using serial to parallel convertor MCP23S17.

```
program SerialGLCD

include __Lib_Glcd_Images

dim
  ii as byte
  jj as word
  someText as string[ 20]

sub procedure delay2S
  delay_ms(2000)
end sub

main:
  ADPCFG = 0xFFFF

  ' for faster spi communication use Spi_Init_Advanced routine
  Spi1_Init()
  Spi_Glcd_Init()
  Spi_Glcd_Fill(0xAA)
  delay2S

  while TRUE
    Spi_Glcd_Fill(0x00)
    Spi_Glcd_Image( truck_bmp )
    delay2S

    for jj = 1 to 40
      Spi_Glcd_Dot(jj,jj,1)
    next jj

    delay2S

    Spi_Glcd_Fill(0x00)
    Spi_Glcd_Line(120, 1, 5,60, 1)
    delay2S
    Spi_Glcd_Line(12, 42, 5,60, 1)
    delay2S

    Spi_Glcd_Rectangle(12, 20, 93,57, 1)
    delay2S

    ' continues...
```

```
    ' continues...

    Spi_Glcd_Line(120, 12, 12,60, 1)
    delay2S

    Spi_Glcd_H_Line(5, 15, 6, 1)
    Spi_Glcd_Line(0, 12, 120, 60, 1)
    Spi_Glcd_V_Line(7, 63, 127, 1)
    delay2S

    for ii = 1 to 10
      Spi_Glcd_Circle(63, 32, 3*ii, 1)
    next ii

    delay2S
    Spi_Glcd_Box(12, 20, 70, 57, 2)
    delay2S

    Spi_Glcd_Fill(0x00)

    Spi_Glcd_Set_Font(@System3x6, 3, 6, 32)
    someText = "SMALL FONT: 3X6"
    Spi_Glcd_Write_Text(someText, 20, 5, 1)

    Spi_Glcd_Set_Font(@FontSystem5x8, 5, 8, 32)
    someText = "Large Font 5x8"
    Spi_Glcd_Write_Text(someText, 3, 4, 1)
    delay2S
  wend
end.
```

Hardware Connection



SPI LCD Library (4-bit interface)

mikroBasic for dsPIC30/33 and PIC24 provides a library for communication with LCD (with HD44780 compliant controllers) in 4-bit mode via SPI interface.

For creating a custom set of LCD characters use the LCD Custom Character Tool.

Note: When using this library with the dsPIC33 and PIC24 family MCUs be aware of their voltage incompatibility with a certain number of the LCD modules. So, additional external power supply for these modules may be required.

Note: The library uses the SPI module for communication. The user must initialize the appropriate SPI module before using the SPI LCD Library. For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `Spi_Set_Active()` function. See the Spi Library functions.

Note: This Library is designed to work with the mikroElektronika's Serial LCD Adapter Board pinout. See schematic at the bottom of this page for details.

Library Routines

```
Spi_Lcd_Config
Spi_Lcd_Init
Spi_Lcd_Out
Spi_Lcd_Out_Cp
Spi_Lcd_Chr
Spi_Lcd_Chr_Cp
Spi_Lcd_Cmd
```

Spi_Lcd_Config

Prototype	`sub procedure Spi_Lcd_Config(dim DeviceAddress as byte, dim byref rstport as word, dim rstpin as word, dim byref csport as word, dim cspin as word)`
Description	Initializes the LCD module via SPI interface. Parameters : - `DeviceAddress`: spi expander hardware address, see schematic at the bottom of this page - `rstport`: reset signal port address - `rstpin`: reset signal pin - `csport`: chip select signal port address - `cspin`: chip select signal pin
Requires	The SPI module needs to be initialized. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.

Example	`' initialize the SPI1 module and LCD module`
	`Spi_Init()`
	`Spi_Lcd_Config(0, PORTF, 0, PORTF, 1)`

Spi_Lcd_Init

Prototype	`sub procedure Spi_Lcd_Init()`
Description	Initializes the LCD module via SPI interface with the `DeviceAddress=0`, `reset` signal connected to the RF0 pin and the chip `select` signal connected to the RF1 pin. **Note**: The `Spi_Lcd_Init()` routine can be changed by the user. Source code is located in the setup library file in the Uses folder.
Requires	The SPI module needs to be initialized. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.

Example	`' initialize the SPI1 module and LCD module`
	`Spi_Init()`
	`Spi_Lcd_Init()`

Spi_Lcd_Out

Prototype	`sub procedure Spi_Lcd_Out(dim row, column as byte, dim byref text as string[20])`
Description	Prints text on the LCD starting from specified position. Both string variables and literals can be passed as a text. Parameters : - `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Requires	LCD needs to be initialized for SPI communication, see the Spi_Lcd_Config and Spi_Lcd_Init routines.
Example	`' Write text "Hello!" on LCD starting from row 1, column 3:` `Spi_Lcd_Out(1, 3, "Hello!")`

Spi_Lcd_Out_Cp

Prototype	`sub procedure Spi_Lcd_Out_CP(dim byref text as string[40])`
Description	Prints text on the LCD at current cursor position. Both string variables and literals can be passed as a text. Parameters : - `text`: text to be written
Requires	LCD needs to be initialized for SPI communication, see the Spi_Lcd_Config and Spi_Lcd_Init routines.
Example	`' Write text "Here!" at current cursor position:` `Spi_Lcd_Out_CP("Here!")`

Spi_Lcd_Chr

Prototype	`sub procedure Spi_Lcd_Chr(dim Row, Column, Out_Char as byte)`
Description	Prints character on LCD at specified position. Both variables and literals can be passed as character. Parameters : - `Row`: writing position row number - `Column`: writing position column number - `Out_Char`: character to be written
Requires	LCD needs to be initialized for SPI communication, see the Spi_Lcd_Config and Spi_Lcd_Init routines.
Example	`' Write character "i" at row 2, column 3:` `Spi_Lcd_Chr(2, 3, "i")`

Spi_Lcd_Chr_Cp

Prototype	`sub procedure Spi_Lcd_Chr_CP(dim Out_Char as byte)`
Description	Prints character on LCD at current cursor position. Both variables and literals can be passed as character. Parameters : - `Out_Char`: character to be written
Requires	LCD needs to be initialized for SPI communication, see the Spi_Lcd_Config and Spi_Lcd_Init routines.
Example	`' Write character "e" at current cursor position:` `Spi_Lcd_Chr_Cp("e")`

Spi_Lcd_Cmd

Prototype	`sub procedure Spi_Lcd_Cmd(dim out_char as byte)`
Description	Sends command to LCD. Parameters : - `out_char`: command to be sent **Note**: Predefined constants can be passed to the function, see Available LCD Commands.
Requires	LCD needs to be initialized for SPI communication, see the Spi_Lcd_Config and Spi_Lcd_Init routines.
Example	`' Clear LCD display:` `Spi_Lcd_Cmd(LCD_CLEAR)`

Library Example (default pin settings)

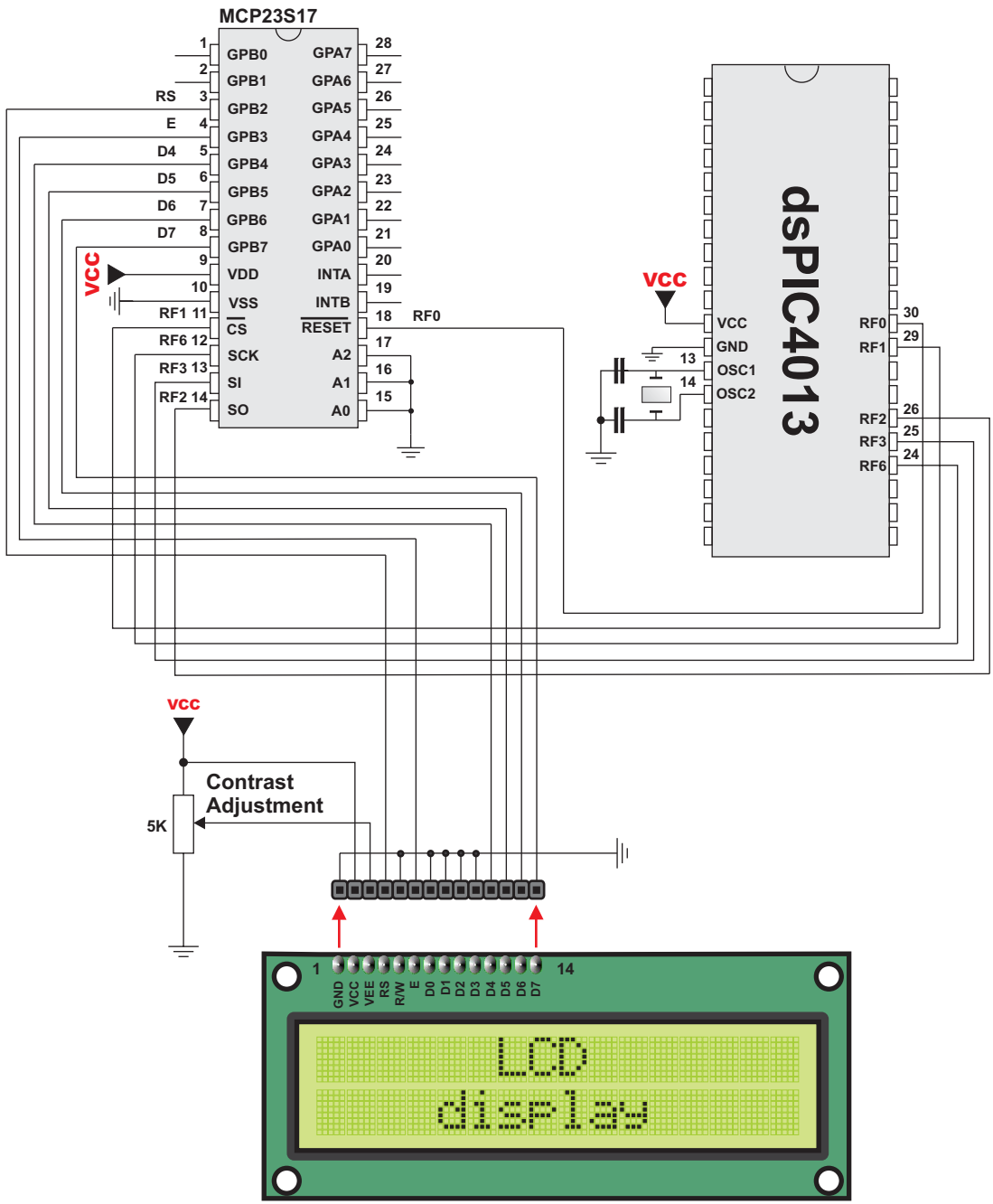
Use Spi_Lcd_Init for default pin settings (see the first figure below).

```
program Spi_Lcd_demo

main:
  ' for faster spi communication use Spi_Init_Advanced routine
  Spi1_Init()                              '  initialize SPI communication
  Spi_Lcd_Init()                          '  initialize lcd via SPI interface
  Spi_Lcd_Cmd(Lcd_CLEAR)                  '  send command to LCD "clear display"
  Spi_Lcd_Cmd(Lcd_CURSOR_OFF)            '  send command cursor off
  Spi_Lcd_Out(2, 1, "mikroElektronika")  ' print txt to LCD, 2nd row, 1st column
  Spi_Lcd_Out(1, 5, "mikroE")          '  print txt to LCD, 1st row, 5th column
  Spi_Lcd_Out(3, 1, "mikroE")
  '  print txt to LCD, 3rd row, 1st column, for lcd's with more than two rows
  Spi_Lcd_Out(4, 10, "mikroE")
  '  print txt to LCD, 4th row, 10th column, for lcd's with more than two rows
end.
```

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

Hardware Connection



SPI LCD8 (8-bit interface) Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for communication with LCD (with HD44780 compliant controllers) in 8-bit mode via SPI interface.

For creating a custom set of LCD characters use the LCD Custom Character Tool.

Note: When using this library with the dsPIC33 and PIC24 family MCUs be aware of their voltage incompatibility with a certain number of LCD modules. So, additional external power supply for these modules may be required.

Note: Library uses the SPI module for communication. The user must initialize the appropriate SPI module before using the SPI LCD Library. For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `Spi_Set_Active()` function. See the Spi Library functions.

Note: This Library is designed to work with mikroElektronika's Serial LCD/GLCD Adapter Board pinout, see schematic at the bottom of this page for details.

Library Routines

```
Spi_Lcd8_Config
Spi_Lcd8_Init
Spi_Lcd8_Out
Spi_Lcd8_Out_Cp
Spi_Lcd8_Chr
Spi_Lcd8_Chr_Cp
Spi_Lcd8_Cmd
```

Spi_Lcd8_Config

Prototype	`sub procedure Spi_LCD8_Config(dim DeviceAddress as byte, dim byref rstport as word, dim rstpin as word, dim byref csport as word, dim cspin as word)`
Description	Initializes the LCD module via SPI interface. Parameters : - `DeviceAddress`: spi expander hardware address, see schematic at the bottom of this page - `rstport`: reset signal port address - `rstpin`: reset signal pin - `csport`: chip select signal port address - `cspin`: chip select signal pin
Requires	The SPI module needs to be initialized. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.

Example	`' initialize the SPI1 module and`
	`' LCD module in 8-bit mode`
	`Spi_Init()`
	`Spi_Lcd8_Config(0, PORTF, 0, PORTF, 1)`

Spi_Lcd8_Init

Prototype	`sub procedure Spi_Lcd8_Init()`
Description	Initializes the LCD module via SPI interface with the `DeviceAddress=0`, `reset` signal connected to the `RF0` pin and the `chip select` signal connected to the `RF1` pin. **Note**: `Spi_Lcd8_Init()` routine can be changed by the user. Source code is located in setup library file in the Uses folder.
Requires	The SPI module needs to be initialized. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.

Example	`' Initialize the SPI1 module and LCD module in 8-bit mode`
	`Spi_Init()`
	`Spi_Lcd8_Init()`

Spi_Lcd8_Out

Prototype	`sub procedure Spi_LCD8_Out(dim row, column as byte, dim byref text as string[20])`
Description	Prints text on LCD starting from specified position. Both string variables and literals can be passed as a text. Parameters : - `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Requires	LCD needs to be initialized for SPI communication, see the Spi_Lcd8_Config and Spi_Lcd8_Init routines.
Example	`' Write text "Hello!" on LCD starting from row 1, column 3:` `Spi_Lcd8_Out(1, 3, "Hello!")`

Spi_Lcd8_Out_Cp

Prototype	`sub procedure Spi_LCD8_Out_CP(dim byref text as string[20])`
Description	Prints text on LCD at current cursor position. Both string variables and literals can be passed as a text. Parameters : - `text`: text to be written
Requires	LCD needs to be initialized for SPI communication, see the Spi_Lcd8_Config and Spi_Lcd8_Init routines.
Example	`' Write text "Here!" at current cursor position:` `Spi_Lcd8_Out_Cp("Here!")`

Spi_Lcd8_Chr

Prototype	`sub procedure Spi_LCD8_Chr(dim row, column, out_char as byte)`
Description	Prints character on LCD at specified position. Both variables and literals can be passed as character. Parameters : - `row`: writing position row number - `column`: writing position column number - `out_char`: character to be written
Requires	LCD needs to be initialized for SPI communication, see the Spi_Lcd8_Config and Spi_Lcd8_Init routines.
Example	`' Write character "i" at row 2, column 3:` `Spi_Lcd8_Chr(2, 3, "i")`

Spi_Lcd8_Chr_Cp

Prototype	`sub procedure Spi_LCD8_Chr_CP(dim out_char as byte)`
Description	Prints character on LCD at current cursor position. Both variables and literals can be passed as character. Parameters : - `out_char`: character to be written
Requires	LCD needs to be initialized for SPI communication, see the Spi_Lcd8_Config and Spi_Lcd8_Init routines.
Example	`' Write character "e" at current cursor position:` `Spi_Lcd8_Chr_Cp("e")`

Spi_Lcd8_Cmd

Prototype	`sub procedure Spi_LCD8_Cmd(dim out_char as byte)`
Description	Sends command to LCD. Parameters : - `out_char`: command to be sent **Note**: Predefined constants can be passed to the function, see Available LCD Commands.
Requires	LCD needs to be initialized for SPI communication, see the Spi_Lcd8_Config and Spi_Lcd8_Init routines.
Example	`' Clear LCD display:` `Spi_Lcd8_Cmd(LCD_CLEAR)`

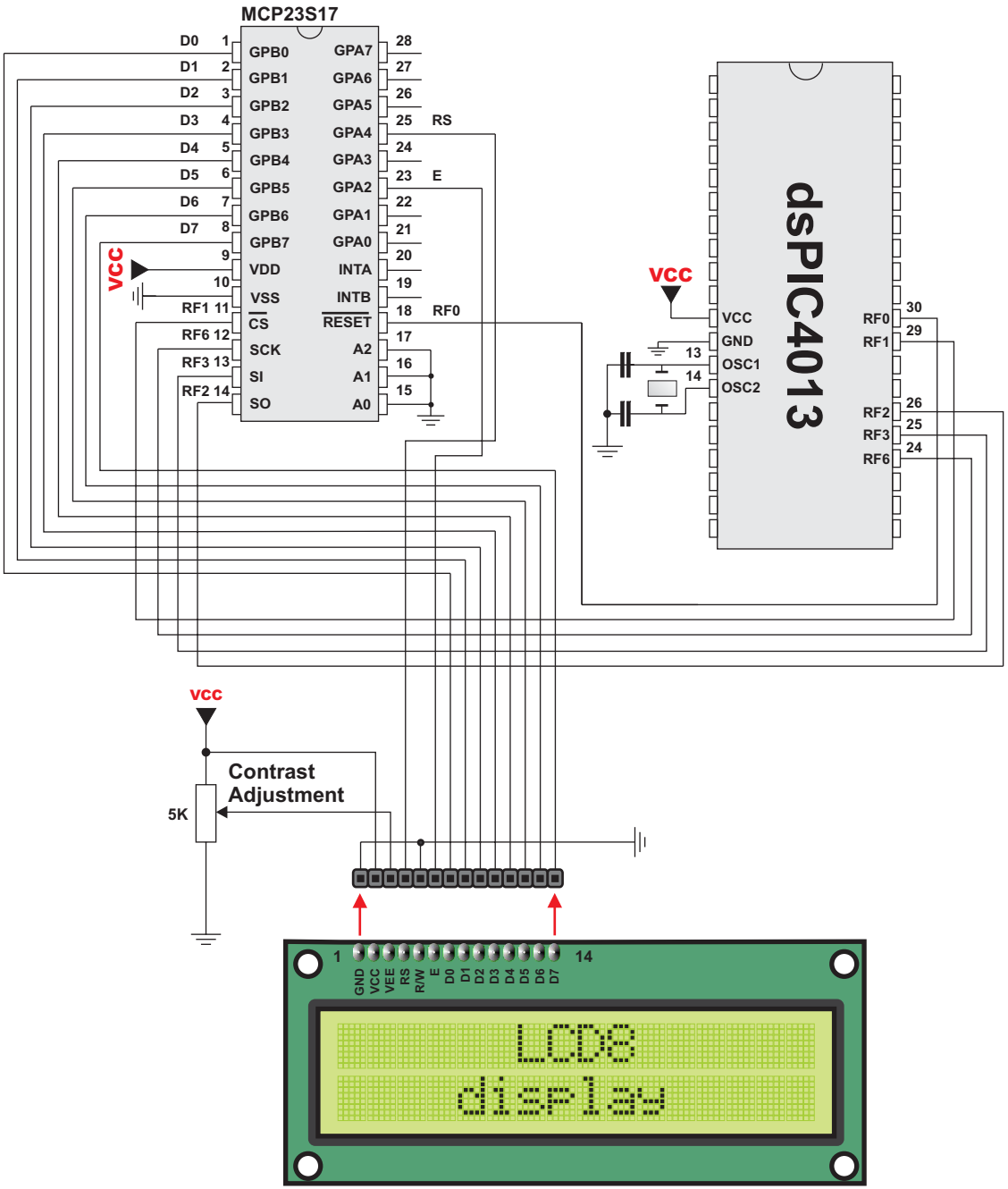
Library Example (default pin settings)

```
program SpiLcd8

main:
    ' for faster spi communication use Spi_Init_Advanced routine
    Spi1_Init()                                  ' initialize SPI communication
    Spi_Lcd8_Init()                              ' initialize lcd via SPI interface
    Spi_Lcd8_Cmd(LCD_CLEAR)                      ' Clear display
    Spi_Lcd8_Cmd(LCD_CURSOR_OFF)                 ' Turn cursor OFF
    Spi_Lcd8_Out(1, 5, "mikroE")                 ' Write Txt to LCD
    Spi_Lcd8_Out(2, 1, "mikroElektronika")  ' Write Txt to LCD
    Spi_Lcd8_Out(3, 1, "mikroE")                 ' Write Txt to LCD
    Spi_Lcd8_Out(4, 10, "mikroE")                ' Write Txt to LCD
end.
```

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

Hardware Connection



SPI T6963C Graphic LCD Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for working with GLCDs based on TOSHIBA T6963C controller via SPI interface. The Toshiba T6963C is a very popular LCD controller for the use in small graphics modules. It is capable of controlling displays with a resolution up to 240x128. Because of its low power and small outline it is most suitable for mobile applications such as PDAs, MP3 players or mobile measurement equipment. Although this controller is small, it has a capability of displaying and merging text and graphics and it manages all interfacing signals to the displays Row and Column drivers.

For creating a custom set of GLCD images use the GLCD Bitmap Editor Tool.

Note: When using this library with the dsPIC33 and PIC24 family MCUs be aware of their voltage incompatibility with a certain number of T6963C based GLCD modules. So, additional external power supply for these modules may be required.

Note: GLCD size based initialization routines can be found in setup library files located in the Uses folder. The user must make sure that used MCU has appropriate ports and pins. If this is not the case the, user should adjust initialization routines.

Note: The library uses the SPI module for communication. The user must initialize the appropriate SPI module before using the Spi T6963C GLCD Library. For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the Spi_Set_Active() function. See the Spi Library functions.

Note: This Library is designed to work with mikroElektronika's Serial GLCD 240x128 and 240x64 Adapter Boards pinout, see schematic at the bottom of this page for details.

Note: Some mikroElektronika's adapter boards have pinout different from T6369C datasheets. Appropriate relations between these labels are given in the table below:

Adapter Board	T6369C datasheet
RS	C/D
R/W	/RD
E	/WR

Library Routines

```
Spi_T6963C_Init
Spi_T6963C_WriteData
Spi_T6963C_WriteCommand
Spi_T6963C_SetPtr
Spi_T6963C_WaitReady
Spi_T6963C_Fill
Spi_T6963C_Dot
Spi_T6963C_Write_Char
Spi_T6963C_Write_Text
Spi_T6963C_Line
Spi_T6963C_Rectangle
Spi_T6963C_Box
Spi_T6963C_Circle
Spi_T6963C_Image
Spi_T6963C_Sprite
Spi_T6963C_Set_Cursor
Spi_T6963C_ClearBit
Spi_T6963C_SetBit
Spi_T6963C_NegBit
Spi_T6963C_DisplayGrPanel
Spi_T6963C_DisplayTxtPanel
Spi_T6963C_SetGrPanel
Spi_T6963C_SetTxtPanel
Spi_T6963C_PanelFill
Spi_T6963C_GrFill
Spi_T6963C_TxtFill
Spi_T6963C_Cursor_Height
Spi_T6963C_Graphics
Spi_T6963C_Text
Spi_T6963C_Cursor
Spi_T6963C_Cursor_Blink
```

Spi_T6963C_Init

Prototype

```
sub procedure Spi_T6963C_Init(dim width, height, fntW as word,
dim DeviceAddress as byte, dim byref rstport as word, dim rstpin
as word, dim byref csport as word, dim cspin as word, dim wr, rd,
cd, rst as byte)
```

Description

Initalizes the Graphic Lcd controller.

Parameters :

- `width`: width of the GLCD panel
- `height`: height of the GLCD panel
- `fntW`: font width
- `DeviceAddress`: spi expander hardware address, see schematic at the bottom of this page
- `rstport`: port expander's reset signal port address
- `rstpin`: port expander's reset signal pin
- `csport`: port expander's chip select signal port address
- `cspin`: port expander's chip select signal pin
- `wr`: write signal pin on GLCD control port
- `rd`: read signal pin on GLCD control port
- `cd`: cd signal pin on GLCD control port
- `rst`: rst signal pin on GLCD control port

Display RAM organization:
The library cuts RAM into panels : a complete panel is one graphics panel followed by a text panel (see schematic below).

<table>
<tr><td>GRAPHICS PANEL #0</td><td rowspan="2">PANEL 0</td></tr>
<tr><td>TEXT PANEL #0</td></tr>
<tr><td>GRAPHICS PANEL #1</td><td rowspan="2">PANEL 1</td></tr>
<tr><td>TEXT PANEL #2</td></tr>
</table>

' continues on the next page ...

' continued from the previous page ...	
Requires	The SPI module needs to be initialized. See the Spi1_Init, Spi1_Init_Advanced, Spi2_Init, Spi2_Init_Advanced, Spi_Init and Spi_Init_Advanced routines.

Example	`Spi1_Init()`
	`Spi_T6963C_Init(240, 64, 8, PORTF, 0, PORTF, 1, 0, 1, 3, 4, 0)`
	`' init display for 240 pixel width and 64 pixel height`
	`' 8 bits character width`
	`' reset pin on PORTB.1`
	`' chip select pin on PORTB.0`
	`' bit 0 is !WR`
	`' bit 1 is !RD`
	`' bit 3 is !CD`
	`' bit 4 is RST`
	`' chip enable, reverse on, 8x8 font internaly set in library`
	`' device address is 0`

Spi_T6963C_writeData

Prototype	`sub procedure Spi_T6963C_WriteData(dim data as byte)`
Description	Writes data to T6963C controller via SPI interface. Parameters : - `data`: data to be written
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_WriteData(AddrL)`

Spi_T6963C_writeCommand

Prototype	`sub procedure Spi_T6963C_WriteCommand(dim data as byte)`
Description	Writes command to T6963C controller via SPI interface. Parameters : - `data`: command to be written
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_WriteCommand(T6963C_CURSOR_POINTER_SET)`

Spi_T6963C_setPtr

Prototype	`sub procedure Spi_T6963C_SetPtr(dim p as word, dim c as byte)`
Description	Sets the memory pointer p for command c. Parameters : - `p`: address where command should be written - `c`: command to be written
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_SetPtr(T6963C_grHomeAddr + start, T6963C_ADDRESS_POINTER_SET)`

Spi_T6963C_waitReady

Prototype	`sub procedure Spi_T6963C_WaitReady()`
Description	Pools the status byte, and loops until Toshiba GLCD module is ready.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_WaitReady()`

Spi_T6963C_fill

Prototype	`sub procedure Spi_T6963C_Fill(dim v as byte, dim start, len as word)`
Description	Fills controller memory block with given byte. Parameters : - `v`: byte to be written - `start`: starting address of the memory block - `len`: length of the memory block in bytes
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_Fill(0x33,0x00FF,0x000F)`

Spi_T6963C_dot

Prototype	`sub procedure Spi_T6963C_Dot(dim x, y as integer, dim color as byte)`
Description	Draws a dot in the current graphic panel of GLCD at coordinates (x, y). Parameters : - `x`: dot position on x-axis - `y`: dot position on y-axis - `color`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the `Spi_T6963C_Init` routine.
Example	`Spi_T6963C_Dot(x0, y0, T6963C_BLACK)`

Spi_T6963C_write_char

Prototype	`sub procedure Spi_T6963C_Write_Char(dim c, x, y, mode as byte)`
Description	Writes a char in the current text panel of GLCD at coordinates (x, y). Parameters : - `c`: char to be written - `x`: char position on x-axis - `y`: char position on y-axis - `mode`: mode parameter. Valid values: `T6963C_ROM_MODE_OR`, `T6963C_ROM_MODE_XOR`, `T6963C_ROM_MODE_AND` and `T6963C_ROM_MODE_TEXT` Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically "OR-ed". This is the most common way of combining text and graphics, for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical "exclusive OR". This can be useful to display text in negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical "AND function". - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_Write_Char("A",22,23,T6963C_ROM_MODE_AND)`

Spi_T6963C_write_text

Prototype	`sub procedure Spi_T6963C_Write_Text(dim byref str as byte[10], dim x, y, mode as byte)`
Description	Writes text in the current text panel of GLCD at coordinates (x, y). Parameters : - `str`: text to be written - `x`: text position on x-axis - `y`: text position on y-axis - `mode`: mode parameter. Valid values: `T6963C_ROM_MODE_OR`, `T6963C_ROM_MODE_XOR`, `T6963C_ROM_MODE_AND` and `T6963C_ROM_MODE_TEXT` Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically "OR-ed". This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical "exclusive OR". This can be useful to display text in negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on the display are combined via the logical "AND function". - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_Write_Text("GLCD LIBRARY DEMO, WELCOME !", 0, 0, T6963C_ROM_MODE_XOR)`

Spi_T6963C_line

Prototype	`sub procedure Spi_T6963C_Line(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	Draws a line from (x0, y0) to (x1, y1). Parameters : - `x0`: x coordinate of the line start - `y0`: y coordinate of the line end - `x1`: x coordinate of the line start - `y1`: y coordinate of the line end - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_Line(0, 0, 239, 127, T6963C_WHITE)`

Spi_T6963C_rectangle

Prototype	`sub procedure Spi_T6963C_Rectangle(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	Draws a rectangle on GLCD. Parameters : - `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_Rectangle(20, 20, 219, 107, T6963C_WHITE)`

Spi_T6963C_box

Prototype	`sub procedure Spi_T6963C_Box(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	Draws a box on the GLCD Parameters : - `x0`: x coordinate of the upper left box corner - `y0`: y coordinate of the upper left box corner - `x1`: x coordinate of the lower right box corner - `y1`: y coordinate of the lower right box corner - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_Box(0, 119, 239, 127, T6963C_WHITE)`

Spi_T6963C_circle

Prototype	`sub procedure Spi_T6963C_Circle(dim x, y as integer, dim r as longint, dim pcolor as word)`
Description	Draws a circle on the GLCD. Parameters : - `x`: x coordinate of the circle center - `y`: y coordinate of the circle center - `r`: radius size - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_Circle(120, 64, 110, T6963C_WHITE)`

Spi_T6963C_image

Prototype	`sub procedure Spi_T6963C_Image(dim const pic as ^byte)`
Description	Displays bitmap on GLCD. Parameters : - `pic`: image to be displayed. Bitmap array can be located in both code and RAM (due to the mikroBasic for dsPIC30 pointer to const and pointer to RAM equivalency). Use the mikroBasic's integrated GLCD Bitmap Editor (menu option **Tools › GLCD Bitmap Editor**) to convert image to a constant array suitable for displaying on GLCD.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_Image(my_image)`

Spi_T6963C_sprite

Prototype	`sub procedure Spi_T6963C_Sprite(dim px, py as byte, dim const pic as ^byte, dim sx, sy as byte)`
Description	Fills graphic rectangle area (px, py) to (px+sx, py+sy) with custom size picture. Parameters : - `px`: x coordinate of the upper left picture corner - `py`: y coordinate of the upper left picture corner - `pic`: picture to be displayed - `sx`: picture width - `sy`: picture height
Requires	Toshiba GLCD module needs to be initialized. See the `Spi_T6963C_Init` routine.
Example	`Spi_T6963C_Sprite(76, 4, einstein, 88, 119) ' draw a sprite`

Spi_T6963C_set_cursor

Prototype	`sub procedure Spi_T6963C_Set_Cursor(dim x, y as byte)`
Description	Sets cursor to row x and column y. Parameters : - x: cursor position row number - y: cursor position column number
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_Set_Cursor(cposx, cposy)`

Spi_T6963C_clearBit

Prototype	`sub procedure Spi_T6963C_ClearBit(dim b as byte)`
Description	Clears control port bit(s). Parameters : - b: bit mask. The function will clear bit x on control port if bit x in bit mask is set to 1.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' clear bits 0 and 1 on control port` `Spi_T6963C_ClearBit(0x0003)`

Spi_T6963C_setBit

Prototype	`sub procedure Spi_T6963C_SetBit(dim b as byte)`
Description	Sets control port bit(s). Parameters : - b: bit mask. The function will set bit x on control port if bit x in bit mask is set to 1.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' set bits 0 and 1 on control port` `Spi_T6963C_SetBit(0x0003)`

Spi_T6963C_negBit

Prototype	`sub procedure Spi_T6963C_NegBit(dim b as byte)`
Description	Negates control port bit(s). Parameters : - `b`: bit mask.The function will negate bit x on control port if bit x in bit mask is set to 1.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' negate bits 0 and 1 on control port` `Spi_T6963C_NegBit(0x0003)`

Spi_T6963C_displayGrPanel

Prototype	`sub procedure Spi_T6963C_DisplayGrPanel(dim n as word)`
Description	Display selected graphic panel. Parameters : - `n`: graphic panel number. Valid values: 0 and 1.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' display graphic panel 1` `Spi_T6963C_DisplayGrPanel(1)`

Spi_T6963C_displayTxtPanel

Prototype	`sub procedure Spi_T6963C_DisplayTxtPanel(dim n as word)`
Description	Display selected text panel. Parameters : - `n`: text panel number. Valid values: 0 and 1.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' display text panel 1` `Spi_T6963C_DisplayTxtPanel(1)`

Spi_T6963C_setGrPanel

Prototype	`sub procedure Spi_T6963C_SetGrPanel(dim n as word)`
Description	Compute start address for selected graphic panel and set appropriate internal pointers. All subsequent graphic operations will be preformed at this graphic panel. Parameters : - n: graphic panel number. Valid values: 0 and 1
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' set graphic panel 1 as current graphic panel.` `Spi_T6963C_SetGrPanel(1)`

Spi_T6963C_setTxtPanel

Prototype	`sub procedure Spi_T6963C_SetTxtPanel(dim n as word)`
Description	Compute start address for selected text panel and set appropriate internal pointers. All subsequent text operations will be preformed at this text panel. Parameters : - n: text panel number. Valid values: 0 and 1.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' set text panel 1 as current text panel.` `Spi_T6963C_SetTxtPanel(1)`

Spi_T6963C_panelFill

Prototype	`sub procedure Spi_T6963C_PanelFill(dim v as word)`
Description	Fill current panel in full (graphic+text) with appropriate value (0 to clear). Parameters : - v: value to fill panel with.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' clear current panel` `Spi_T6963C_PanelFill(0)`

Spi_T6963C_grFill

Prototype	`sub procedure Spi_T6963C_GrFill(dim v as word)`
Description	Fill current graphic panel with appropriate value (0 to clear). Parameters : - `v`: value to fill graphic panel with.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' clear current graphic panel` `Spi_T6963C_GrFill(0)`

Spi_T6963C_txtFill

Prototype	`sub procedure Spi_T6963C_TxtFill(dim v as word)`
Description	Fill current text panel with appropriate value (0 to clear). Parameters : - `v`: this value increased by 32 will be used to fill text panel.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' clear current text panel` `Spi_T6963C_TxtFill(0)`

Spi_T6963C_cursor_height

Prototype	`sub procedure Spi_T6963C_Cursor_Height(dim n as byte)`
Description	Set cursor size. Parameters : - `n`: cursor height. Valid values: 0..7.
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`Spi_T6963C_Cursor_Height(7)`

Spi_T6963C_graphics

Prototype	`sub procedure Spi_T6963C_Graphics(dim n as word)`
Description	Enable/disable graphic displaying. Parameters : - n: graphic enable/disable parameter. Valid values: 0 (disable graphic dispaying) and 1 (enable graphic displaying).
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' enable graphic displaying` `Spi_T6963C_Graphics(1)`

Spi_T6963C_text

Prototype	`sub procedure Spi_T6963C_Text(dim n as word)`
Description	Enable/disable text displaying. Parameters : - n: text enable/disable parameter. Valid values: 0 (disable text dispaying) and 1 (enable text displaying).
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' enable text displaying` `Spi_T6963C_Text(1)`

Spi_T6963C_cursor

Prototype	`sub procedure Spi_T6963C_Cursor(dim n as word)`
Description	Set cursor on/off. Parameters : - n: on/off parameter. Valid values: 0 (set cursor off) and 1 (set cursor on).
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' set cursor on` `Spi_T6963C_Cursor(1)`

Spi_T6963C_cursor_blink

Prototype	`sub procedure Spi_T6963C_Cursor_Blink(dim n as word)`
Description	Enable/disable cursor blinking. Parameters : - `n`: cursor blinking enable/disable parameter. Valid values: 0 (disable cursor blinking) and 1 (enable cursor blinking).
Requires	Toshiba GLCD module needs to be initialized. See the Spi_T6963C_Init routine.
Example	`' enable cursor blinking` `Spi_T6963C_Cursor_Blink(1)`

Library Example

The following drawing demo tests advanced routines of the Spi T6963C GLCD library.

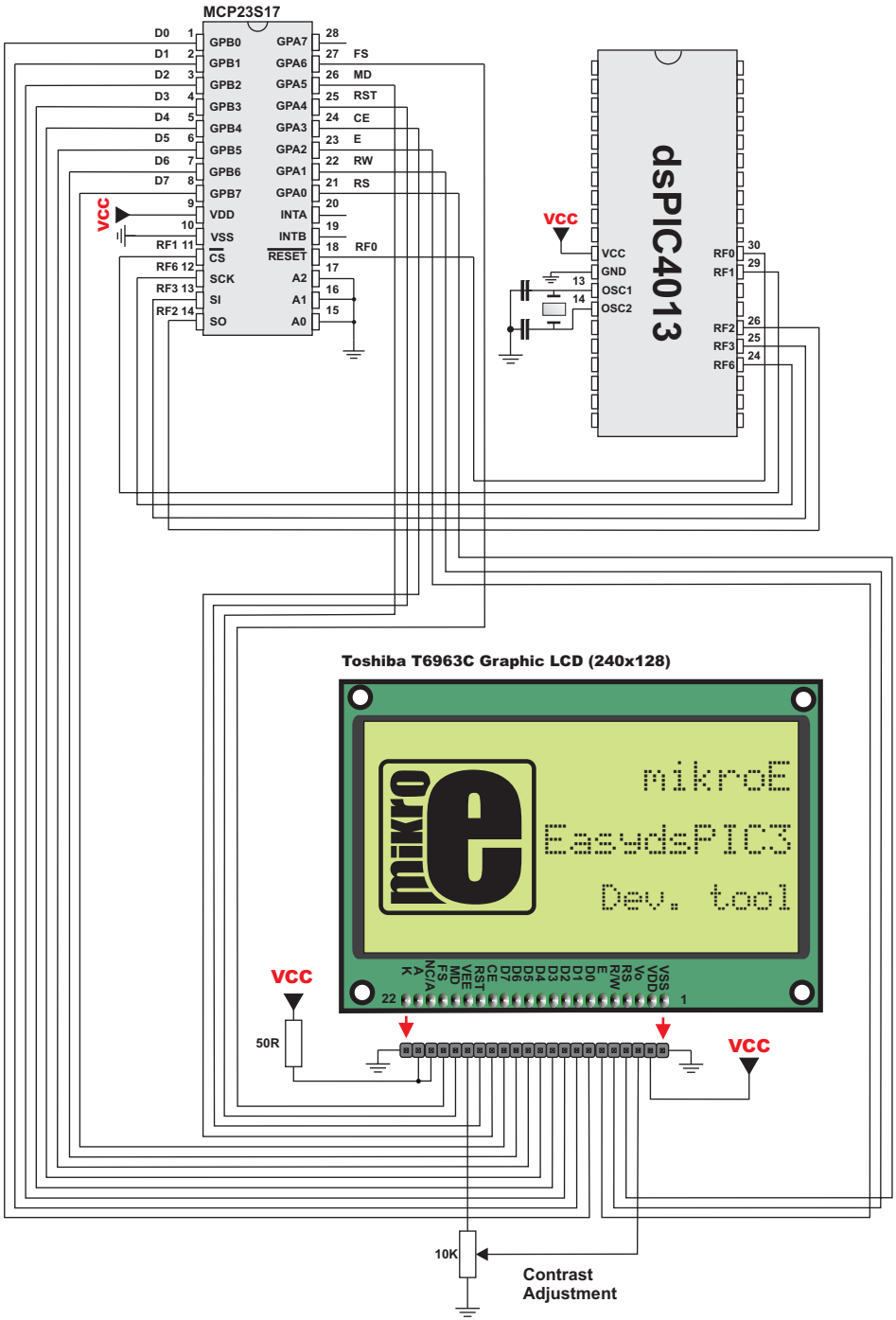
```
program Spi_T6963C_240x128

include __Lib_T6963CConst
include bitmap
include bitmap2

dim    panel as byte          '* current panel
            i as word          '* general purpose register
         curs as byte          '* cursor visibility
       cposx,
       cposy as word           '* cursor x-y position
       txtcols as byte         '* number of text colons

main:
  ADPCFG = 0xFFFF
  PORTB  = 0
  TRISB  = 0xFF


' continues ...
```

```
  ' continued ...

  '* init display for 240 pixel width and 128 pixel height
  '* 8 bits character width
  '* data bus on MCP23S17 portB
  '* control bus on MCP23S17 portA
  '* bit 2 is !WR
  '* bit 1 is !RD
  '* bit 0 is !CD
  '* bit 4 is RST
  '* chip enable, reverse on, 8x8 font internaly set in library
  ' for faster spi communication use Spi_Init_Advanced routine
  Spi1_Init()
  Spi_T6963C_Init_240x128()
  Spi_T6963C_panelFill(0)

  '* enable both graphics and text display at the same time

  Spi_T6963C_graphics(1)
  Spi_T6963C_text(1)

  panel = 0
  i = 0
  curs = 0
  cposy = 0
  cposx = 0
  txtcols = 240 div 8     '* calculate number of text colomns
                        '* (grafic display width divided by font width)
  '* text messages

  Spi_T6963C_write_text(" GLCD LIBRARY DEMO, WELCOME !", 0, 0,
T6963C_ROM_MODE_XOR)
  Spi_T6963C_write_text(" EINSTEIN WOULD HAVE LIKED mE", 0, 15,
T6963C_ROM_MODE_XOR)

  '* cursor

  Spi_T6963C_cursor_height(8)          '* 8 pixel height
  Spi_T6963C_set_cursor(0, 0)          '* move cursor to top left
  Spi_T6963C_cursor(0)                  '* cursor off

  '* draw rectangles

  Spi_T6963C_rectangle(0, 0, 239, 127, T6963C_WHITE)
  Spi_T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE)
  Spi_T6963C_rectangle(40, 40, 199, 87, T6963C_WHITE)
  Spi_T6963C_rectangle(60, 60, 179, 67, T6963C_WHITE)

  ' continues ...
```

```
' continued ...

'* draw a cross

Spi_T6963C_line(0, 0, 239, 127, T6963C_WHITE)
Spi_T6963C_line(0, 127, 239, 0, T6963C_WHITE)

'* draw solid boxes

Spi_T6963C_box(0, 0, 239, 8, T6963C_WHITE)
Spi_T6963C_box(0, 119, 239, 127, T6963C_WHITE)

'* draw cicles

Spi_T6963C_circle(120, 64, 10, T6963C_WHITE)
Spi_T6963C_circle(120, 64, 30, T6963C_WHITE)
Spi_T6963C_circle(120, 64, 50, T6963C_WHITE)
Spi_T6963C_circle(120, 64, 70, T6963C_WHITE)
Spi_T6963C_circle(120, 64, 90, T6963C_WHITE)
Spi_T6963C_circle(120, 64, 110, T6963C_WHITE)
Spi_T6963C_circle(120, 64, 130, T6963C_WHITE)

Spi_T6963C_sprite(76, 4, einstein, 88, 119)    '* draw a sprite

Spi_T6963C_setGrPanel(1)   '* select other graphic panel

Spi_T6963C_image(mikrobasic_glcd_bmp)
'* fill the graphic screen with a picture

while true
    '*
    '* if RB1 is pressed, toggle the display between graphic
    '* panel 0 and graphic 1
    '*
     if(PORTB.1 <> 0) then
       panel = panel + 1
       panel = panel and 1
       Spi_T6963C_displayGrPanel(panel)
       Delay_ms(300)

   '* if RB2 is pressed, display only graphic panel
   '*
     else
       if(PORTB.2 <> 0) then
         Spi_T6963C_graphics(1)
         Spi_T6963C_text(0)
         Delay_ms(300)


' continues ...
```

```
' continued ...
      else '* if RB3 is pressed, display only text panel
        if(PORTB.3 <> 0) then
          Spi_T6963C_graphics(0)
          Spi_T6963C_text(1)
          Delay_ms(300)
       else'* if RB4 is pressed, display text and graphic panels
        if(PORTB.4 <> 0) then
          Spi_T6963C_graphics(1)
          Spi_T6963C_text(1)
          Delay_ms(300)
        else '* if RB5 is pressed, change cursor
          if(PORTB.5 <> 0) then
            curs = curs + 1
            if(curs = 3) then
              curs = 0
            end if
            select case curs
              case 0
                    Spi_T6963C_cursor(0)

              case 1
                    Spi_T6963C_cursor(1)
                    Spi_T6963C_cursor_blink(1)

              case 2
                    Spi_T6963C_cursor(1)
                    Spi_T6963C_cursor_blink(0)
            end select
            Delay_ms(300)
          end if
        end if
      end if
    end if
  end if
  '* move cursor, even if not visible
  cposx = cposx + 1
  if(cposx = txtcols) then
    cposx = 0
    cposy = cposy + 1
    if(cposy = (128 div T6963C_CHARACTER_HEIGHT)) then
    '* if y end
      cposy = 0   '* grafic height (128) div character height
    end if
  end if
  Spi_T6963C_set_cursor(cposx, cposy)
  Delay_ms(100)
wend
end.
```

Hardware Connection



T6963C Graphic LCD Library

mikroBasic for dsPIC30/33 and PIC24 provides a library for working with GLCDs based on TOSHIBA T6963C controller. The Toshiba T6963C is a very popular LCD controller for the use in small graphics modules. It is capable of controlling displays with a resolution up to 240x128. Because of its low power and small outline it is most suitable for mobile applications such as PDAs, MP3 players or mobile measurement equipment. Although small, this contoller has a capability of displaying and merging text and graphics and it manages all the interfacing signals to the displays Row and Column drivers.

For creating a custom set of GLCD images use the GLCD Bitmap Editor Tool.

Note: When using this library with the dsPIC33 and PIC24 family MCUs be aware of their voltage incompatibility with a certain number of T6963C based GLCD modules. So, additional external power supply for these modules may be required.

Note: ChipEnable(CE), FontSelect(FS) and Reverse(MD) have to be set to appropriate levels by the user outside of the T6963C_Init() function. See the Library Example code at the bottom of this page.

Note: GLCD size based initialization routines can be found in setup library files located in the Uses folder. The user must make sure that used MCU has appropriate ports and pins. If this is not the case the user should adjust initialization routines.

Note: Some mikroElektronika's adapter boards have pinout different from T6369C datasheets. Appropriate relations between these labels are given in the table below:

Adapter Board	T6369C datasheet
RS	C/D
R/W	/RD
E	/WR

Library Routines

```
T6963C_Init
T6963C_WriteData
T6963C_WriteCommand
T6963C_SetPtr
T6963C_WaitReady
T6963C_Fill
T6963C_Dot
T6963C_Write_Char
T6963C_Write_Text
T6963C_Line
T6963C_Rectangle
T6963C_Box
T6963C_Circle
T6963C_Image
T6963C_Sprite
T6963C_Set_Cursor
T6963C_ClearBit
T6963C_SetBit
T6963C_NegBit
T6963C_DisplayGrPanel
T6963C_DisplayTxtPanel
T6963C_SetGrPanel
T6963C_SetTxtPanel
T6963C_PanelFill
T6963C_GrFill
T6963C_TxtFill
T6963C_Cursor_Height
T6963C_Graphics
T6963C_Text
T6963C_Cursor
T6963C_Cursor_Blink
```

T6963C_Init

Prototype	`sub procedure T6963C_Init(dim width, height, fntW as word, dim byref data as word, dim byref cntrl as word, dim wr, rd, cd, rst as word)`

Description	Initalizes the Graphic Lcd controller.
	
	Parameters :
	
	- `width`: width of the GLCD panel
	- `height`: height of the GLCD panel
	- `fntW`: font width
	- `data`: data PORT
	- `cntrl`: control PORT
	- `wr`: write signal pin
	- `rd`: read signal pin
	- `cd`: command/data signal pin
	- `rst`: reset signal pin
	
	Display RAM organization:
	The library cuts the RAM into panels : a complete panel is one graphics panel followed by a text panel (see schematic below).
	
	`GRAPHICS PANEL #0` / `TEXT PANEL #0` — `PANEL 0`
	`GRAPHICS PANEL #1` / `TEXT PANEL #2` — `PANEL 1`
	
	' continues on the next page ...

' continued from the previous page ...	

Example	`T6963C_Init(240, 128, 8, PORTF, PORTD, 2, 1, 0, 4)`
	`' init display for 240 pixel width and 128 pixel height`
	`' 8 bits character width`
	`' data bus on PORTF`
	`' control bus on PORTD`
	`' bit 2 is !WR`
	`' bit 1 is !RD`
	`' bit 0 is !CD`
	`' bit 4 is RST`

T6963C_writeData

Prototype	`sub procedure T6963C_WriteData(dim data as byte)`
Description	Writes data to T6963C controller. Parameters : - `data`: data to be written
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_WriteData(AddrL)`

T6963C_writeCommand

Prototype	`sub procedure T6963C_WriteCommand(dim data as byte)`
Description	Writes command to the T6963C controller. Parameters : - `data`: command to be written
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_WriteCommand(T6963C_CURSOR_POINTER_SET)`

T6963C_setPtr

Prototype	`sub procedure T6963C_SetPtr(dim p as word, dim c as byte)`
Description	Sets the memory pointer p for command c. Parameters : - `p`: address where command should be written - `c`: command to be written
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_SetPtr(T6963C_grHomeAddr + start, T6963C_ADDRESS_POINTER_SET)`

T6963C_waitReady

Prototype	`sub procedure T6963C_WaitReady()`
Description	Pools the status byte, and loops until Toshiba GLCD module is ready.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_WaitReady()`

T6963C_fill

Prototype	`sub procedure T6963C_Fill(dim v as byte, dim start, len as word)`
Description	Fills controller memory block with given byte. Parameters : - `v`: byte to be written - `start`: starting address of the memory block - `len`: length of the memory block in bytes
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Fill(0x33,0x00FF,0x000F)`

T6963C_dot

Prototype	`sub procedure T6963C_Dot(dim x, y as integer, dim color as byte)`
Description	Draws a dot in the current graphic panel of GLCD at coordinates (x, y). Parameters : - `x`: dot position on x-axis - `y`: dot position on y-axis - `color`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Dot(x0, y0, pcolor)`

T6963C_write_char

Prototype	`sub procedure T6963C_Write_Char(dim c, x, y, mode as byte)`
Description	Writes a char in the current text panel of GLCD at coordinates (x, y). Parameters : - `c`: char to be written - `x`: char position on x-axis - `y`: char position on y-axis - `mode`: mode parameter. Valid values: `T6963C_ROM_MODE_OR`, `T6963C_ROM_MODE_XOR`, `T6963C_ROM_MODE_AND` and `T6963C_ROM_MODE_TEXT` Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically "OR-ed". This is the most common way of combining text and graphics, for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical "exclusive OR". This can be useful to display text in the negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical "AND function". - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Write_Char("A",22,23,AND)`

T6963C_write_text

Prototype	`sub procedure T6963C_Write_Text(dim byref str as byte[10], dim x, y, mode as byte)`
Description	Writes text in the current text panel of GLCD at coordinates (x, y). Parameters : - `str`: text to be written - `x`: text position on x-axis - `y`: text position on y-axis - `mode`: mode parameter. Valid values: `T6963C_ROM_MODE_OR`, `T6963C_ROM_MODE_XOR`, `T6963C_ROM_MODE_AND` and `T6963C_ROM_MODE_TEXT` Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically "OR-ed". This is the most common way of combining text and graphics, for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical "exclusive OR". This can be useful to display text in the negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical "AND function". - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Write_Text("GLCD LIBRARY DEMO, WELCOME !", 0, 0, T6963C_ROM_MODE_XOR)`

T6963C_line

Prototype	`sub procedure T6963C_Line(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	Draws a line from (x0, y0) to (x1, y1). Parameters : - `x0`: x coordinate of the line start - `y0`: y coordinate of the line end - `x1`: x coordinate of the line start - `y1`: y coordinate of the line end - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Line(0, 0, 239, 127, T6963C_WHITE)`

T6963C_rectangle

Prototype	`sub procedure T6963C_Rectangle(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	Draws a rectangle on GLCD. Parameters : - `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Rectangle(20, 20, 219, 107, T6963C_WHITE)`

T6963C_box

Prototype	`sub procedure T6963C_Box(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	Draws a box on GLCD Parameters : - `x0`: x coordinate of the upper left box corner - `y0`: y coordinate of the upper left box corner - `x1`: x coordinate of the lower right box corner - `y1`: y coordinate of the lower right box corner - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Box(0, 119, 239, 127, T6963C_WHITE)`

T6963C_circle

Prototype	`sub procedure T6963C_Circle(dim x, y as integer, dim r as longint, dim pcolor as word)`
Description	Draws a circle on GLCD. Parameters : - `x`: x coordinate of the circle center - `y`: y coordinate of the circle center - `r`: radius size - `pcolor`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Circle(120, 64, 110, T6963C_WHITE)`

T6963C_image

Prototype	`sub procedure T6963C_Image(dim const pic as ^byte)`
Description	Displays bitmap on GLCD. Parameters : - `pic`: image to be displayed. Bitmap array can be located in both code and RAM (due to the mikroBasic for dsPIC30 pointer to const and pointer to RAM equivalency). Use the mikroBasic's integrated GLCD Bitmap Editor (menu option **Tools › GLCD Bitmap Editor**) to convert image to a constant array suitable for displaying on GLCD.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Image(my_image)`

T6963C_sprite

Prototype	`sub procedure T6963C_Sprite(dim px, py as byte, dim const pic as ^byte, dim sx, sy as byte)`
Description	Fills graphic rectangle area (px, py) to (px+sx, py+sy) with custom size picture. Parameters : - `px`: x coordinate of the upper left picture corner - `py`: y coordinate of the upper left picture corner - `pic`: picture to be displayed - `sx`: picture width - `sy`: picture height
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Sprite(76, 4, einstein, 88, 119) ' draw a sprite`

T6963C_set_cursor

Prototype	`sub procedure T6963C_Set_Cursor(dim x, y as byte)`
Description	Sets cursor to row x and column y. Parameters : - x: cursor position row number - y: cursor position column number
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Set_Cursor(cposx, cposy)`

T6963C_clearBit

Prototype	`sub procedure T6963C_ClearBit(dim b as word)`
Description	Clears control port bit(s). Parameters : - b: bit mask.The function will clear bit x on control port if bit x in bit mask is set to 1.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' clear bits 0 and 1 on control port` `T6963C_ClearBit(0x0003)`

T6963C_setBit

Prototype	`sub procedure T6963C_SetBit(dim b as word)`
Description	Sets control port bit(s). Parameters : - b: bit mask. The function will set bit x on control port if bit x in bit mask is set to 1.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' set bits 0 and 1 on control port` `T6963C_SetBit(0x0003)`

T6963C_negBit

Prototype	`sub procedure T6963C_NegBit(dim b as word)`
Description	Negates control port bit(s). Parameters : - `b`: bit mask.The function will negate bit x on control port if bit x in bit mask is set to 1.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' negate bits 0 and 1 on control port` `T6963C_NegBit(0x0003)`

T6963C_displayGrPanel

Prototype	`sub procedure T6963C_DisplayGrPanel(dim n as word)`
Description	Display selected graphic panel. Parameters : - `n`: graphic panel number. Valid values: 0 and 1.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' display graphic panel 1` `T6963C_DisplayGrPanel(1)`

T6963C_displayTxtPanel

Prototype	`sub procedure T6963C_DisplayTxtPanel(dim n as word)`
Description	Display selected text panel. Parameters : - `n`: text panel number. Valid values: 0 and 1.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' display text panel 1` `T6963C_DisplayTxtPanel(1)`

T6963C_setGrPanel

Prototype	`sub procedure T6963C_SetGrPanel(dim n as word)`
Description	Compute start address for selected graphic panel and set appropriate internal pointers. All subsequent graphic operations will be preformed at this graphic panel. Parameters : - `n`: graphic panel number. Valid values: 0 and 1.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' set graphic panel 1 as current graphic panel.` `T6963C_SetGrPanel(1)`

T6963C_setTxtPanel

Prototype	`sub procedure T6963C_SetTxtPanel(dim n as word)`
Description	Compute start address for selected text panel and set appropriate internal pointers. All subsequent text operations will be preformed at this text panel. Parameters : - `n`: text panel number. Valid values: 0 and 1.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' set text panel 1 as current text panel.` `T6963C_SetTxtPanel(1)`

T6963C_panelFill

Prototype	`sub procedure T6963C_PanelFill(dim v as word)`
Description	Fill current panel in full (graphic+text) with appropriate value (0 to clear). Parameters : - `v`: value to fill panel with.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' clear current panel` `T6963C_PanelFill(0)`

T6963C_grFill

Prototype	`sub procedure T6963C_GrFill(dim v as word)`
Description	Fill current graphic panel with appropriate value (0 to clear). Parameters : - `v`: value to fill graphic panel with.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' clear current graphic panel` `T6963C_GrFill(0)`

T6963C_txtFill

Prototype	`sub procedure T6963C_TxtFill(dim v as word)`
Description	Fill current text panel with appropriate value (0 to clear). Parameters : - `v`: this value increased by 32 will be used to fill text panel.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' clear current text panel` `T6963C_TxtFill(0)`

T6963C_cursor_height

Prototype	`sub procedure T6963C_Cursor_Height(dim n as word)`
Description	Set cursor size. Parameters : - `n`: cursor height. Valid values: 0..7.
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Cursor_Height(7)`

T6963C_graphics

Prototype	`sub procedure T6963C_Graphics(dim n as word)`
Description	Enable/disable graphic displaying. Parameters : - `n`: on/off parameter. Valid values: 0 (disable graphic dispaying) and 1 (enable graphic displaying).
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' enable graphic displaying` `T6963C_Graphics(1)`

T6963C_text

Prototype	`sub procedure T6963C_Text(dim n as word)`
Description	Enable/disable text displaying. Parameters : - `n`: on/off parameter. Valid values: 0 (disable text dispaying) and 1 (enable text displaying).
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' enable text displaying` `T6963C_Text(1)`

T6963C_cursor

Prototype	`sub procedure T6963C_Cursor(dim n as word)`
Description	Set cursor on/off. Parameters : - `n`: on/off parameter. Valid values: 0 (set cursor off) and 1 (set cursor on).
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' set cursor on` `T6963C_Cursor(1)`

T6963C_cursor_blink

Prototype	`sub procedure T6963C_Cursor_Blink(dim n as word)`
Description	Enable/disable cursor blinking. Parameters : - n: on/off parameter. Valid values: 0 (disable cursor blinking) and 1 (enable cursor blinking).
Requires	Toshiba GLCD module needs to be initialized. See the T6963C_Init routine.
Example	`' enable cursor blinking` `T6963C_Cursor_Blink(1)`

Library Example

The following drawing demo tests advanced routines of T6963C GLCD library.

```
program T6963C

include __Lib_T6963CConst
include bitmap
include bitmap2

dim   panel as byte          ' current panel
          i as word          ' general purpose register
       curs as byte          ' cursor visibility
      cposx,
      cposy as word          ' cursor x-y position

main:
  ADPCFG = 0xFFFF
  PORTD  = 0
  TRISD  = 0xFF
  TRISB  = 0                ' portb is output only
  PORTB  = %00000000        ' chip enable, reverse on, 8x8 font

  T6963C_init_240x128()
  'T6963C_init(240, 128, 8, PORTB, PORTF, 2, 1, 0, 4)


' continues...
```

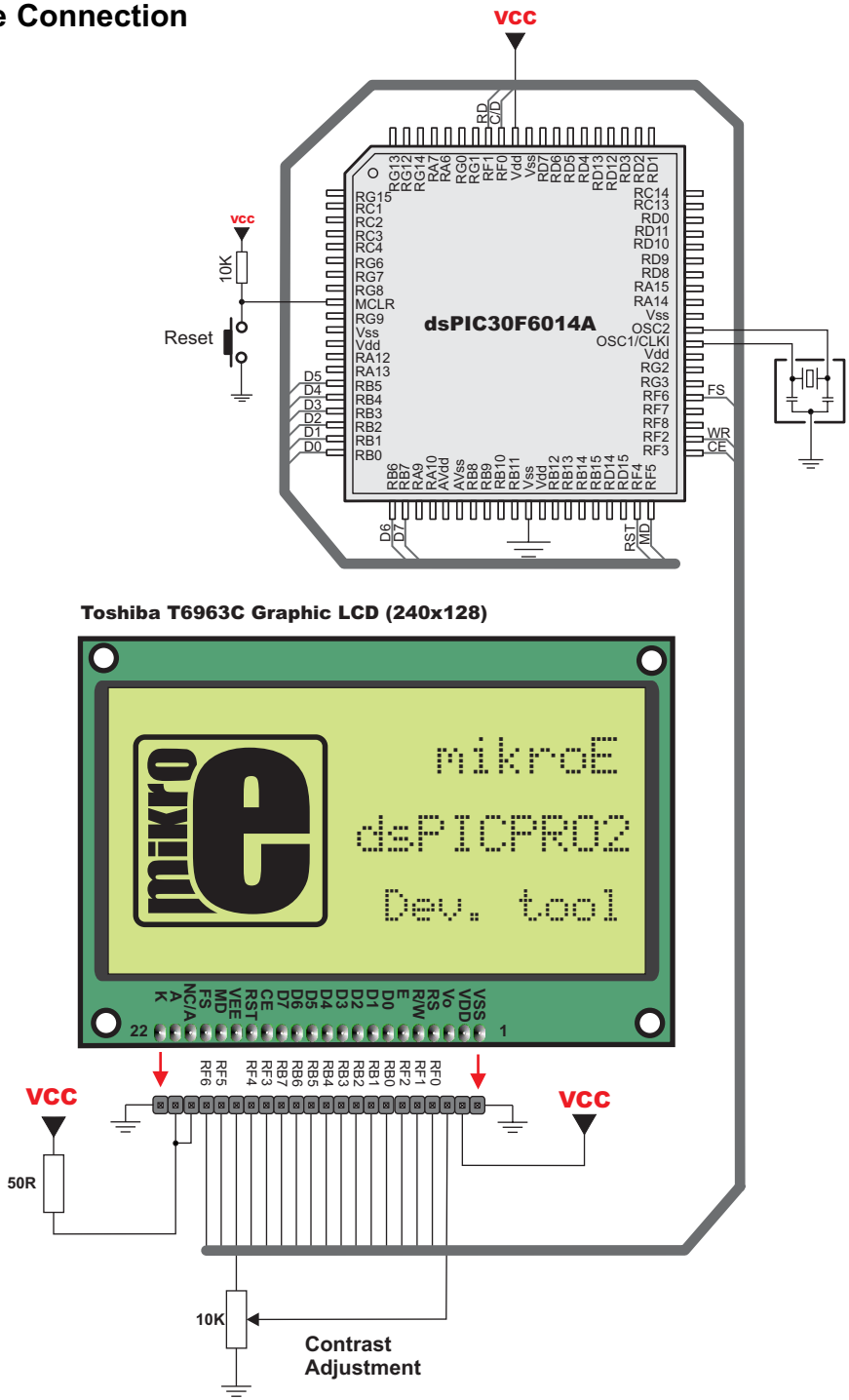
```
' continued...

  '* enable both graphics and text display at the same time

  T6963C_graphics(1)
  T6963C_text(1)

  panel = 0
  i = 0
  curs = 0
  cposy = 0
  cposx = 0

  '* text messages

  T6963C_write_text(" GLCD LIBRARY DEMO, WELCOME !", 0, 0,
T6963C_ROM_MODE_XOR)
  T6963C_write_text(" EINSTEIN WOULD HAVE LIKED mE", 0, 15,
T6963C_ROM_MODE_XOR)

  '* cursor

  T6963C_cursor_height(8)        ' 8 pixel height
  T6963C_set_cursor(0, 0)        ' move cursor to top left
  T6963C_cursor(0)                ' cursor off


  '* draw rectangles

  T6963C_rectangle(0, 0, 239, 127, T6963C_WHITE)
  T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE)
  T6963C_rectangle(40, 40, 199, 87, T6963C_WHITE)
  T6963C_rectangle(60, 60, 179, 67, T6963C_WHITE)


  '* draw a cross

  T6963C_line(0, 0, 239, 127, T6963C_WHITE)
  T6963C_line(0, 127, 239, 0, T6963C_WHITE)


  '* draw solid boxes

  T6963C_box(0, 0, 239, 8, T6963C_WHITE)
  T6963C_box(0, 119, 239, 127, T6963C_WHITE)



' continues...
```

```
' continued...

  '*
  '* draw circles
  '*
  T6963C_circle(120, 64, 10, T6963C_WHITE)
  T6963C_circle(120, 64, 30, T6963C_WHITE)
  T6963C_circle(120, 64, 50, T6963C_WHITE)
  T6963C_circle(120, 64, 70, T6963C_WHITE)
  T6963C_circle(120, 64, 90, T6963C_WHITE)
  T6963C_circle(120, 64, 110, T6963C_WHITE)
  T6963C_circle(120, 64, 130, T6963C_WHITE)

  T6963C_sprite(76, 4, einstein, 88, 119) ' draw a sprite

  T6963C_setGrPanel(1)        ' select other graphic panel

  T6963C_image(mikrobasic_glcd_bmp)
  ' fill the graphic screen with a picture

  while TRUE
    '* if RD0 is pressed, toggle the display
    '* between graphic panel 0 and graphic 1

      if(PORTD.0 <> 0) then
        panel = panel + 1
        panel = panel and 1
        T6963C_displayGrPanel(panel)
        Delay_ms(300)

    '* if RD1 is pressed, display only graphic panel

      else
        if(PORTD.1 <> 0) then
          T6963C_graphics(1)
          T6963C_text(0)
          Delay_ms(300)

    '* if RD2 is pressed, display only text panel

        else
          if(PORTD.2 <> 0) then
            T6963C_graphics(0)
            T6963C_text(1)
            Delay_ms(300)


' continues...
```

```
' continued...

    '*
    '* if RD3 is pressed, display text and graphic panels
    '*
        else
          if(PORTD.3 <> 0) then
            T6963C_graphics(1)
            T6963C_text(1)
            Delay_ms(300)

    '*
    '* if RD4 is pressed, change cursor
    '*
          else
            if(PORTD.4 <> 0) then
              curs = curs + 1
              if(curs = 3) then
                curs = 0
              end if
              select case curs
                case 0
                     T6963C_cursor(0)

                case 1
                     T6963C_cursor(1)
                     T6963C_cursor_blink(1)

                case 2
                     T6963C_cursor(1)
                     T6963C_cursor_blink(0)
              end select
              Delay_ms(300)
            end if
          end if
        end if
      end if
    end if


' continues...
```

```
' continued...

    '*
    '* move cursor, even if not visible
    '*

    cposx = cposx + 1
    if(cposx = T6963C_txtCols) then
      cposx = 0
      cposy = cposy + 1
      if(cposy = (T6963C_grHeight div T6963C_CHARACTER_HEIGHT)) then
        cposy = 0
      end if
    end if
    T6963C_set_cursor(cposx, cposy)
    Delay_ms(100)
  wend
end.
```

Hardware Connection



UART Library

The UART hardware module is available with a number of the dsPIC30/33 and PIC24 MCUs. mikroBasic for dsPIC30/33 and PIC24 UART Library provides comfortable work with the Asynchronous (full duplex) mode.

Note: For the dsPIC30/33 and PIC24 MCUs with the multiple UART modules there are the UART1 (supports UART1 module), UART2 (supports UART2 module) and UART (supports both UART modules) libraries. Switching between the UART modules in the UART library is done by the `Uart_Set_Active` function (both UART modules have to be previously initialized).

Library Routines

```
Uart1_Init
Uart1_Init_Advanced
Uart1_Data_Ready
Uart1_Read_Char
Uart1_Read_Text
Uart1_Write_Char
Uart1_Write_Text
```

```
Uart2_Init
Uart2_Init_Advanced
Uart2_Data_Ready
Uart2_Read_Char
Uart2_Read_Text
Uart2_Write_Char
Uart2_Write_Text
```

```
Uart_Init
Uart_Init_Advanced
Uart_Data_Ready
Uart_Read_Char
Uart_Read_Text
Uart_Write_Char
Uart_Write_Text
Uart_Set_Active
```

Uart1_Init

Prototype	`sub procedure Uart1_Init(dim baud_rate as longint)`
Description	Configures and initializes the UART1 module. The internal UART1 module module is set to: - continue operation in IDLE mode - default Tx and Rx pins - loopback mode disabled - 8-bit data, no parity - 1 STOP bit - transmitter enabled - generate interrupt on transmission end - interrupt on reception enabled - Address Detect mode disabled Parameters : - `baud_rate`: requested baud rate Refer to the device data sheet for baud rates allowed for specific Fosc. **Note**: For the dsPIC33 and pic24 MCUs, the compiler will choose at which speed the calculation is to be performed (high or low). This does not mean that it is the best choice for desired baud rate. If the baud rate error generated in this way is too big then the Uart1_Init_Advanced routine, which allows speed selection, should be used.
Requires	MCU with the UART1 module.

Example	`' Initialize hardware UART and establish communication`
	`' at 2400 bps`
	`Uart1_Init(2400)`

Uart1_Init_Advanced

Prototype (dsPIC30)	`sub procedure Uart1_Init_Advanced(dim baud_rate as longint, dim parity, stop_bits as word)`
Prototype (dsPIC33 and pic24)	`sub procedure Uart1_Init_Advanced(dim baud_rate as longint, dim parity, stop_bits as word, dim high_low_speed as word)`
Description	Configures and initializes the UART1 module. Parameters : - `baud_rate`: requested baud rate - `parity`: parity and data selection parameter. Valid values: 6 : 9-bit data, no parity 4 : 8-bit data, odd parity 2 : 8-bit data, even parity 0 : 8-bit data, no parity - `stop_bits`: stop bit selection parameter. Valid values: 1 : 2 STOP bits 0 : 1 STOP bit - `high_low_speed`: high/low speed selection parameter. Available only for dsPIC33 and pic24 MCUs. Valid values: 0 : use low speed calculations 1 : use high speed calculations - `other` : let the compiler decide which speed (low or high) to use. This does not mean that it is the best choice for desired baud rate Refer to the device data sheet for baud rates allowed for specific *Fosc*.
Requires	MCU with the UART1 module.

Example	`' dsPIC30 family example`
	`' Initialize hardware UART and establish communication at`
	`' 2400 bps, 8-bit data, even parity and 2 STOP bits`
	`Uart1_Init_Advanced(2400, 2, 1)`
	
	`' dsPIC33 and pic24 family example`
	`' Initialize hardware UART and establish communication at`
	`' 2400 bps, 8-bit data, even parity, 2 STOP bits and high speed`
	`' baud rate calculations`
	`Uart1_Init_Advanced(2400, 2, 1, 1)`

Uart1_Data_Ready

Prototype	`sub function Uart1_Data_Ready() as word`
Returns	- `1` if data is ready for reading - `0` if there is no data in the receive register
Description	The function tests if the data in receive buffer is ready for reading.
Requires	MCU with the UART1 module. The UART1 module must be initialized before using this routine. See the Uart1_Init routine.

Example	`dim receive as word`
	`...`
	`' read data if ready`
	`if (Uart1_Data_Ready() = 1) then`
	`  receive = Uart1_Read_Char()`
	`  ...`

Uart1_Read_Char

Prototype	`sub function Uart1_Read_Char() as word`
Returns	Received byte.
Description	The function receives a byte via UART1. Use the Uart1_Data_Ready function to test if data is ready.
Requires	MCU with the UART1 module. The UART1 module must be initialized before using this routine. See the Uart1_Init routine.

Example	`dim receive as word`
	`...`
	`' read data if ready`
	`if (Uart1_Data_Ready() = 1) then`
	`  receive = Uart1_Read_Char()`
	`  ...`

Uart1_Read_Text

Prototype	`sub procedure Uart1_Read_Text(dim byref output, delimiter as string[20])`
Description	Procedure receives text via UART1. Received bytes are stored into output string until the delimiter sequence is received. The procedure will end when delimiter is received, otherwise the procedure will end after 20 recieved characters. Parameters : - `output`: buffer for storing received data - `delimiter`: delimiter string
Requires	MCU with the UART1 module. The UART1 module must be initialized before using this routine. See the Uart1_Init routine.

Example	`' Read text until the sequence "OK" is received,`
	`' and then send it back:`
	`Uart1_Init(9600)`
	`delim = "OK"`
	`while TRUE`
	`  if Uart1_Data_Ready() = 1 then`
	`  Uart1_Read_Text(txt, delim)`
	`  Uart1_Write_Text(txt)`
	`  end if`
	`wend`

Uart1_Write_Char

Prototype	`sub procedure Uart1_Write_Char(dim data as word)`
Description	The function transmits a byte via the UART1 module. Parameters : - `data`: data to be sent
Requires	MCU with the UART1 module. The UART1 module must be initialized before using this routine. See the Uart1_Init routine.

Example	`dim data as byte`
	`...`
	`data = 0x1E`
	`Uart1_Write_Char(data)`

Uart1_Write_Text

Prototype	`sub procedure Uart1_Write_Text(dim byref uart_text as string[20])`
Description	Procedure sends text via UART1. Parameters : - `uart_text`: text to be sent
Requires	MCU with the UART1 module. The UART1 module must be initialized before using this routine. See the Uart1_Init routine.
Example	`Uart1_Write_Text("Hello world!")  ' send message via UART`

Uart2_Init

Prototype	`sub procedure Uart2_Init(dim baud_rate as longint)`
Description	Configures and initializes the UART2 module. The internal UART2 module module is set to: - continue operation in IDLE mode - default Tx and Rx pins - loopback mode disabled - 8-bit data, no parity - 1 STOP bit - transmitter enabled - generate interrupt on transmission end - interrupt on reception enabled - Address Detect mode disabled Parameters : - `baud_rate`: requested baud rate Refer to the device datasheet for baud rates allowed for specific *Fosc*. **Note**: For the dsPIC33 and pic24 MCUs, the compiler will choose at which speed the calculation is to be performed (high or low). This does not mean that it is the best choice for desired baud rate. If the baud rate error generated in this way is too big then the Uart2_Init_Advanced routine, which allows speed selection, should be used.
Requires	MCU with the UART2 module.

Example	`' Initialize hardware UART and establish communication`
	`' at 2400 bps`
	`Uart2_Init(2400)`

Uart2_Init_Advanced

Prototype (dsPIC30)	`sub procedure Uart2_Init_Advanced(dim baud_rate as longint, dim parity, stop_bits as word)`
Prototype (dsPIC33 and pic24)	`sub procedure Uart2_Init_Advanced(dim baud_rate as longint, dim parity, stop_bits as word, dim high_low_speed as word)`
Description	Configures and initializes the UART2 module. Parameters : - `baud_rate`: requested baud rate - `parity`: parity and data selection parameter. Valid values: 6 : 9-bit data, no parity 4 : 8-bit data, odd parity 2 : 8-bit data, even parity 0 : 8-bit data, no parity - `stop_bits`: stop bit selection parameter. Valid values: 1 : 2 STOP bits 0 : 1 STOP bit - `high_low_speed`: high/low speed selection parameter. Available only for dsPIC33 and pic24 MCUs. Valid values: 0 : use low speed calculations 1 : use high speed calculations - `other` : let the compiler decide which speed (low or high) to use. This does not mean that it is the best choice for desired baud rate Refer to the device data sheet for baud rates allowed for specific *Fosc*.
Requires	MCU with the UART2 module.

Example	`' dsPIC30 family example`
	`' Initialize hardware UART and establish communication at`
	`' 2400 bps, 8-bit data, even parity and 2 STOP bits`
	`Uart2_Init_Advanced(2400, 2, 1)`
	
	`' dsPIC33 and pic24 family example`
	`' Initialize hardware UART and establish communication at`
	`' 2400 bps, 8-bit data, even parity, 2 STOP bits and high speed`
	`' baud rate calculations`
	`Uart2_Init_Advanced(2400, 2, 1, 1)`

Uart2_Data_Ready

Prototype	`sub function Uart2_Data_Ready() as word`
Returns	- 1 if data is ready for reading - 0 if there is no data in the receive register
Description	The function tests if data in receive buffer is ready for reading.
Requires	MCU with the UART2 module. The UART2 module must be initialized before using this routine. See the Uart2_Init routine.

Example	`dim receive as word`
	`...`
	`' read data if ready`
	`if (Uart2_Data_Ready() = 1) then`
	`  receive = Uart2_Read_Char()`
	`  ...`

Uart2_Read_Char

Prototype	`sub function Uart2_Read_Char() as word`
Returns	Received byte.
Description	The function receives a byte via UART2. Use the Uart2_Data_Ready function to test if data is ready.
Requires	MCU with the UART2 module. The UART2 module must be initialized before using this routine. See the Uart2_Init routine.

Example	`dim receive as word`
	`...`
	`' read data if ready`
	`if (Uart2_Data_Ready() = 1) then`
	`  receive = Uart2_Read_Char()`
	`  ...`

Uart2_Read_Text

Prototype	`sub procedure Uart2_Read_Text(dim byref output, delimiter as string[ 20] )`
Description	Procedure receives text via UART2. Received bytes are stored into output string until the delimiter sequence is received. The procedure will end when delimiter is received, otherwise the procedure will end after 20 recieved characters. Parameters : - `output`: buffer for storing received data - `delimiter`: delimiter string
Requires	MCU with the UART2 module. The UART2 module must be initialized before using this routine. See the Uart2_Init routine.

Example	`' Read text until the sequence "OK" is received,`
	`' and then send it back:`
	`Uart2_Init(9600)`
	`delim = "OK"`
	`while TRUE`
	`  if Uart2_Data_Ready() = 1 then`
	`  Uart2_Read_Text(txt, delim)`
	`  Uart2_Write_Text(txt)`
	`  end if`
	`wend`

Uart2_Write_Char

Prototype	`sub procedure Uart2_Write_Char(dim data as word)`
Description	The function transmits a byte via the UART2 module. Parameters : - `data`: data to be sent
Requires	MCU with the UART2 module. The UART2 module must be initialized before using this routine. See the Uart2_Init routine.

Example	`dim data as byte`
	`...`
	`data = 0x1E`
	`Uart2_Write_Char(data)`

Uart2_Write_Text

Prototype	`sub procedure Uart2_Write_Text(dim byref uart_text as string[20])`
Description	Procedure sends text via UART2. Parameters : - `uart_text`: text to be sent
Requires	MCU with the UART2 module. The UART2 module must be initialized before using this routine. See the Uart2_Init routine.
Example	`Uart2_Write_Text("Hello world!")  ' send message via UART`

Uart_Init

Prototype	`sub procedure Uart_Init(dim baud_rate as longint)`
Description	Configures and initializes the UART1 module. The internal UART1 module module is set to: - continue operation in IDLE mode - default Tx and Rx pins - loopback mode disabled - 8-bit data, no parity - 1 STOP bit - transmitter enabled - generate interrupt on transmission end - interrupt on reception enabled - Address Detect mode disabled Parameters : - `baud_rate`: requested baud rate Refer to the device data sheet for baud rates allowed for specific *Fosc*. **Note**: For the dsPIC33 and pic24 MCUs, the compiler will choose at which speed the calculation is to be performed (high or low). This does not mean that it is the best choice for desired baud rate. If the baud rate error generated in this way is too big then the Uart_Init_Advanced routine, which allows speed selection, should be used.
Requires	MCU with the UART1 module.

Example	`' Initialize hardware UART1 and establish communication`
	`' at 2400 bps`
	`Uart_Init(2400)`

Uart_Init_Advanced

Prototype (dsPIC30)	`sub procedure Uart_Init_Advanced(dim baud_rate as longint, dim parity, stop_bits as word)`
Prototype (dsPIC33 and pic24)	`sub procedure Uart_Init_Advanced(dim baud_rate as longint, dim parity, stop_bits as word, dim high_low_speed as word)`
Description	Configures and initializes the UART1 module. Parameters : - `baud_rate`: requested baud rate - `parity`: parity and data selection parameter. Valid values: 6 : 9-bit data, no parity 4 : 8-bit data, odd parity 2 : 8-bit data, even parity 0 : 8-bit data, no parity - `stop_bits`: stop bit selection parameter. Valid values: 1 : 2 STOP bits 0 : 1 STOP bit - `high_low_speed`: high/low speed selection parameter. Available only for dsPIC33 and pic24 MCUs. Valid values: 0 : use low speed calculations 1 : use high speed calculations - `other` : let the compiler decide which speed (low or high) to use. This does not mean that it is the best choice for desired baud rate Refer to the device data sheet for baud rates allowed for specific *Fosc*.
Requires	MCU with the UART1 module.

Example	`' dsPIC30 family example`
	`' Initialize hardware UART and establish communication`
	`' at 2400 bps, 8-bit data, even parity and 2 STOP bits`
	`Uart_Init_Advanced(2400, 2, 1)`
	
	`' dsPIC33 and pic24 family example`
	`' Initialize hardware UART and establish communication`
	`' at 2400 bps, 8-bit data, even parity, 2 STOP bits and high`
	`' speed baud rate calculations`
	`Uart_Init_Advanced(2400, 2, 1, 1)`

Uart_Data_Ready

Prototype	`sub function Uart_Data_Ready() as word`
Returns	- 1 if data is ready for reading - 0 if there is no data in the receive register
Description	The function tests if data in receive buffer is ready for reading. **Note**: if MCU has 2 UART modules, active module will be used. See the Uart_Set_Active routine.
Requires	Routine requires at least one UART module. Used UART module must be initialized before using this function. See the Uart1_Init, Uart1_Init_Advanced, Uart2_Init, Uart2_Init_Advanced, Uart_Init and Uart_Init_Advanced routines.

Example	`dim receive as word`
	`...`
	`' read data if ready`
	`if (Uart_Data_Ready()= 1) then`
	`  receive = Uart_Read_Char()`

Uart_Read_Char

Prototype	`sub function Uart_Read_Char() as word`
Returns	Received byte.
Description	The function receives a byte via active UART module. Use the Uart_Data_Ready function to test if data is ready.
Requires	Routine requires at least one UART module. For choosing the active module see the Uart_Set_Active routine. Used UART module must be initialized before using this function. See the Uart1_Init, Uart1_Init_Advanced, Uart2_Init, Uart2_Init_Advanced, Uart_Init and Uart_Init_Advanced routines.

Example	`dim receive as word`
	`...`
	`' read data if ready`
	`if (Uart_Data_Ready() = 1) then`
	`  receive = Uart_Read_Char()`

Uart_Read_Text

Prototype	`sub procedure Uart_Read_Text(dim byref output, delimiter as string[ 20] )`
Description	Procedure receives text via the active UART module. Received bytes are stored into output string until the delimiter sequence is received. The procedure will end when delimiter is received, otherwise the procedure will end after 20 recieved characters. Parameters : - `output`: buffer for storing received data - `delimiter`: delimiter string
Requires	Routine requires at least one UART module. For choosing the active module see the Uart_Set_Active routine. Used UART module must be initialized before using this function. See the Uart1_Init, Uart1_Init_Advanced, Uart2_Init, Uart2_Init_Advanced, Uart_Init and Uart_Init_Advanced routines.

Example	`' Read text until the sequence "OK" is`
	`' received, and then send it back:`
	`Uart_Init(9600)`
	`delim = "OK"`
	`while TRUE`
	`  if Uart_Data_Ready() = 1 then`
	`  Uart_Read_Text(txt, delim)`
	`  Uart_Write_Text(txt)`
	`  end if`
	`wend`

Uart_Write_Char

Prototype	`sub procedure Uart_Write_Char(dim data as word)`
Description	The function transmits a byte via the active UART module. Parameters : - `data`: data to be sent
Requires	Routine requires at least one UART module. For choosing the active module see the Uart_Set_Active routine. Used UART module must be initialized before using this function. See the Uart1_Init, Uart1_Init_Advanced, Uart2_Init, Uart2_Init_Advanced, Uart_Init and Uart_Init_Advanced routines.

Example	`dim data as byte`
	`...`
	`data = 0x1E`
	`Uart_Write_Char(data)`

Uart_Write_Text

Prototype	`sub procedure Uart_Write_Text(dim byref uart_text as string[20])`
Description	Procedure sends text via active UART module. Parameters : - `uart_text`: text to be sent
Requires	Routine requires at least one UART module. For choosing the active module see the Uart_Set_Active routine. Used UART module must be initialized before using this function. See the Uart1_Init, Uart1_Init_Advanced, Uart2_Init, Uart2_Init_Advanced, Uart_Init and Uart_Init_Advanced routines.
Example	`Uart_Write_Text("Hello world!")  ' send message via UART`

Uart_Set_Active

Prototype	`sub procedure Uart_Set_Active(dim UartNo as byte)`
Description	Sets the active UART module which will be used by Uart_Data_Ready, Uart_Read_Char, Uart_Read_Text, Uart_Write_Char and Uart_Write_Text routines. Parameters : - `UartNo`: module number. Valid values: 1 (for UART1) and 2 (for UART2)
Requires	Routine is available only for MCUs with two UART modules. Used UART module must be initialized before using this function. See the Uart1_Init, Uart1_Init_Advanced, Uart2_Init, Uart2_Init_Advanced, Uart_Init and Uart_Init_Advanced routines.
Example	`Uart_Set_Active(2)  'Sets the UART2 module active`

Library Example

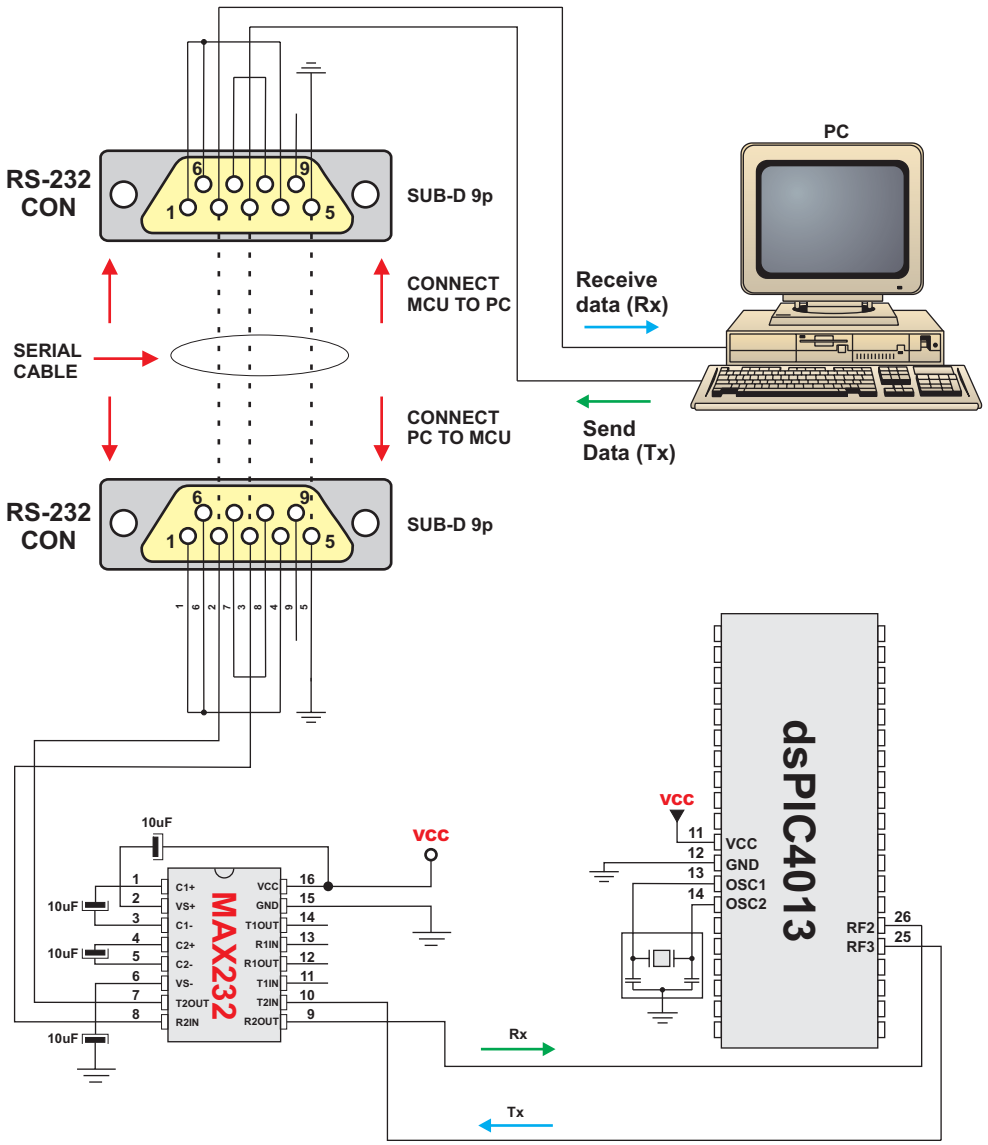
The example demonstrates simple data exchange via UART. When the microcontroller receives data, it immediately sends it back. If dsPIC is connected to the PC (see the figure below), you can test the example from the mikroBasic's terminal for the RS-232 communication, menu item **Tools › USART Terminal**.

```
program Uart1_Test

dim rx1 as word

main:

  Uart1_Init(9600)

  while TRUE
    if Uart1_Data_Ready = 1 then
      rx1 = Uart1_Read_Char()
      Uart1_Write_Char(rx1)
    end if
  wend

end.
```

Here's a simple test for chips with 2 UART modules:

```
program Uart2_Test

dim rx2 as word

main:
  Uart2_Init(9600)
  while true
    if Uart2_Data_Ready = 1 then
      rx2=Uart2_Read_Char
      Uart2_Write_Char(rx2)
    end if
  wend
end.
```

Hardware Connection



Conversions Library

mikroBasic for dsPIC30/33 and PIC24 Conversions Library provides routines for numerals to strings and BCD/decimal conversions.

Library Routines

You can get text representation of numerical value by passing it to one of the following routines:

```
ByteToStr
ShortToStr
WordToStr
WordToStrWithZeros
IntToStr
LongintToStr
LongWordToStr
WordToHex
FloatToStr
StrToInt
StrToWord
```

The following functions convert decimal values to BCD and vice versa:

```
Dec2Bcd
Bcd2Dec16
Dec2Bcd16
```

ByteToStr

Prototype	`sub procedure ByteToStr(dim input as byte, dim byref output as string[ 3] )`
Description	Converts input byte to a string. The output string is right justified and remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: byte to be converted - `output`: destination string

Example	`dim t as byte`
	`  txt as string[ 3]`
	`...`
	`t = 24`
	`ByteToStr(t, txt)  ' txt is " 24" (one blank here)`

ShortToStr

Prototype	`sub procedure ShortToStr(dim input as short, dim byref output as string[ 4] )`
Description	Converts input short (signed byte) number to a string. The output string is right justified and remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: short number to be converted - `output`: destination string

Example	`dim t as short`
	`  txt as string[ 4]`
	`...`
	`t = -24`
	`ByteToStr(t, txt)  ' txt is " -24" (one blank here)`

WordToStr

Prototype	`sub procedure WordToStr(dim input as word, dim byref output as string[ 5] )`
Description	Converts input word to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: word to be converted - `output`: destination string

Example	`dim t as word`
	`  txt as string[ 5]`
	`...`
	`t = 437`
	`WordToStr(t, txt)  ' txt is "  437" (two blanks here)`

WordToStrWithZeros

Prototype	`sub procedure WordToStrWithZeros(dim input as word, dim byref output as string[ 5] )`
Description	Converts input word to a string. The output string is right justified and the remaining positions on the left (if any) are filled with zeros. Parameters : - `input`: word to be converted - `output`: destination string

Example	`dim t as word`
	`  txt as string[ 5]`
	`...`
	`t = 437`
	`WordToStrWithZeros(t, txt)  ' txt is '00437'`

IntToStr

Prototype	`sub procedure IntToStr(dim input as integer, dim byref output as string[ 6] )`
Description	Converts input integer number to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: integer number to be converted - `output`: destination string

Example	`dim input as integer`
	`  txt as string[ 6]`
	`...`
	`input = -4220`
	`IntToStr(input, txt)  ' txt is " -4220"`

LongintToStr

Prototype	`sub procedure LongintToStr(dim input as longint, dim byref output as string[ 11] )`
Description	Converts input longint number to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: longint number to be converted - `output`: destination string

Example	`dim input as longint`
	`  txt as string[ 11]`
	`...`
	`input = -12345678`
	`IntToStr(input, txt)  ' txt is "  -12345678"`

LongWordToStr

Prototype	`sub procedure LongWordToStr(dim input as dword, dim byref output as string[10])`
Description	Converts input double word number to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: double word number to be converted - `output`: destination string

Example	`dim input as longint`
	`  txt as string[10]`
	`...`
	`input = 12345678`
	`IntToStr(input, txt)  ' txt is "  12345678"`

WordToHex

Prototype	`sub procedure WordToHex(dim input as word, dim byref output as string[4])`
Description	Converts input word to hexadecimal digits string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: word to be converted - `output`: destination string

Example	Here's an example which prints hex value of a `word` variable on LCD:
	
	`dim input as word`
	`  txt as string[4]`
	`...`
	`input = 26`
	`WordToHex(input, txt)  ' txt is "001A"`

FloatToStr

Prototype	`sub function FloatToStr(dim fnum as real, dim byref str as string[20]) as byte`
Returns	- 3 if input number is NaN - 2 if input number is -INF - 1 if input number is +INF - 0 if conversion was successful
Description	Converts a floating point number to a string. Parameters : - `fnum`: floating point number to be converted - `str`: destination string The output string is left justified and null terminated after the last digit. **Note**: Given floating point number will be truncated to 7 most significant digits before conversion.

Example	`dim ff1, ff2, ff3 as real`
	`  txt as string[20]`
	`...`
	`ff1 = -374.2`
	`ff2 = 123.456789`
	`ff3 = 0.000001234`
	
	`FloatToStr(ff1, txt)  ' txt is "-374.2"`
	`FloatToStr(ff2, txt)  ' txt is "123.4567"`
	`FloatToStr(ff3, txt)  ' txt is "1.234e-6"`

StrToInt

Prototype	`sub function StrToInt(dim byref input as string[6]) as integer`
Returns	Integer variable.
Description	Converts a string to an integer. Parameters : - `input`: string to be converted
Requires	Input string is assumed to be the correct representation of a number. The conversion will end with the first character which is not a decimal digit.

Example	`dim ii as integer`
	`...`
	`ii= StrToInt("-1234")`

StrToWord

Prototype	`sub function StrToWord(dim byref input as string[6]) as word`
Returns	Word variable.
Description	Converts a string to word. Parameters : - `input`: string to be converted.
Requires	Input string is assumed to be the correct representation of a number. The conversion will end with the first character which is not a decimal digit.

Example	`dim ww as word`
	`...`
	`ww = StrToword("65432")`

Dec2Bcd

Prototype	`sub function Dec2Bcd(dim decnum as byte) as byte`
Returns	Converted BCD value.
Description	Converts input number to its appropriate BCD representation. Parameters : - `decnum`: number to be converted

Example	`dim a, b as byte`
	`...`
	`a = 22`
	`b = Dec2Bcd(a)  ' b equals 34`

Bcd2Dec16

Prototype	`sub function Bcd2Dec16(dim bcdnum as word) as word`
Returns	Converted decimal value.
Description	Converts 16-bit BCD numeral to its decimal equivalent. Parameters : - `bcdnum`: 16-bit BCD numeral to be converted

Example	`dim a, b as word`
	`...`
	`a = 0x1234  ' a equals 4660`
	`b = Bcd2Dec16(a)  ' b equals 1234`

Dec2Bcd16

Prototype	`sub function Dec2Bcd16(dim decnum as word) as word`
Returns	Converted BCD value.
Description	Converts decimal value to its BCD equivalent. Parameters : - `decnum` decimal number to be converted

Example	`dim a, b as word`
	`...`
	`a = 2345`
	`b = Dec2Bcd16(a)  ' b equals 9029`

Math Library

mikroBasic for dsPIC30/33 and PIC24 provides a set of library functions for floating point math handling. See also Predefined Globals and Constants for the list of predefined math constants.

Library Routines

```
acos
asin
atan
atan2
ceil
cos
cosh
exp
fabs
floor
frexp
ldexp
log
log10
modf
pow
sin
sinh
sqrt
tan
tanh
```

acos

Prototype	`sub function acos(dim x as real) as real`
Description	The function returns the arc cosine of parameter x; that is, the value whose cosine is x. The input parameter x must be between -1 and 1 (inclusive). The return value is in radians, between 0 and pi (inclusive).

asin

Prototype	`sub function asin(dim x as real) as real`
Description	The function returns the arc sine of parameter x; that is, the value whose sine is x. The input parameter x must be between -1 and 1 (inclusive). The return value is in radians, between -pi/2 and pi/2 (inclusive).

atan

Prototype	`sub function atan(dim arg as real) as real`
Description	The function computes the arc tangent of parameter arg; that is, the value whose tangent is arg. The return value is in radians, between -pi/2 and pi/2 (inclusive).

atan2

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

ceil

Prototype	`sub function ceil(dim x as real) as real`
Description	The function returns value of parameter x rounded up to the next whole number.

cos

Prototype	`sub function cos(dim arg as real) as real`
Description	The function returns the cosine of arg in radians. The return value is from -1 to 1.

cosh

Prototype	`sub function cosh(dim x as real) as real`
Description	The 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	`sub function exp(dim x as real) as real`
Description	The function returns the value of e — the base of natural logarithms — raised to the power x (i.e. e^x).

fabs

Prototype	`sub function fabs(dim dd as real) as real`
Description	The function returns the absolute (i.e. positive) value of `dd`.

floor

Prototype	`sub function floor(dim x as real) as real`
Description	The function returns the value of parameter x rounded down to the nearest integer.

frexp

Prototype	`sub function frexp(dim byref eptr as integer) as real`
Description	The function splits a floating-point value value into a normalized fraction and an integral power of 2. The return value is a normalized fraction and the integer exponent is stored in the object pointed to by eptr.

ldexp

Prototype	`sub function ldexp(dim value as real, newexp as integer) as real`
Description	The function returns the result of multiplying the floating-point number value by 2 raised to the power newexp (i.e. returns value * 2^{newexp}).

log

Prototype	`sub function log(dim x as real) as real`
Description	The function returns the natural logarithm of x (i.e. $\log_e(x)$).

log10

Prototype	`sub function log10(dim x as real) as real`
Description	Function returns the base-10 logarithm of `x` (i.e. $\log_{10}(x)$).

modf

Prototype	`sub function modf(dim val as real, dim byref iptr as real) as real`
Description	The function returns the signed fractional component of val, placing its whole number component into the variable pointed to by iptr.

pow

Prototype	`sub function pow(dim x, y as real) as real`
Description	The function returns the value of x raised to the power y (i.e. xy). If x is negative, the function will automatically cast y into longint.

sin

Prototype	`sub function sin(dim arg as real) as real`
Description	The function returns the sine of arg in radians. The return value is from -1 to 1.

sinh

Prototype	`sub function sinh(dim x as real) as real`
Description	The 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	`sub function sqrt(dim x as real) as real`
Description	The function returns the non negative square root of x.

tan

Prototype	`sub function tan(dim x as real) as real`
Description	The function returns the tangent of x in radians. The return value spans the allowed range of floating point in mikroBasic for dsPIC30/33 and PIC24.

tanh

Prototype	`sub function tanh(dim x as real) as real`
Description	The function returns the hyperbolic tangent of x, defined mathematically as sinh(x)/cosh(x).

Setjmp Library

The Setjmp library contains functions and types definitions for bypassing the normal function call and return discipline.

Library Routines

```
Setjmp
Longjmp
```

Setjmp

Prototype	`sub function setjmp(dim byref env as word[4]) as integer`
Returns	- `0` if the return is from direct invocation - `nonzero` value if the return is from a call to longjmp (this value will be set by the longjmp routine)
Description	This function saves calling position for a later use by longjmp. Parameters : - `env`: buffer suitable for holding information needed for restoring calling environment

Example	`dim buf as word[4]`
	`...`
	`Setjmp(buf)`

Longjmp

Prototype	`sub procedure longjmp(dim byref env as word[ 4] , dim val as integer)`
Description	Restores calling environment saved in the env buffer by the most recent invocation of setjmp. If there has been no such invocation, or the function containing the invocation of setjmp has terminated in the interim, the behavior is undefined. Parameters : - `env`: buffer holding the information saved by the corresponding setjmp invocation - `val`: value to be returned by the corresponding setjmp function
Requires	Invocation of `longjmp` must occur before return from the function in which setjmp was called encounters.

Example	`dim buf as word[ 4]`
	`...`
	`Longjmp(buf, 2)`

Library Example

Example demonstrates function cross calling using setjmp and longjmp functions. When called, Setjmp() saves its calling environment in its **buf** argument for later use by the Longjmp(). Longjmp(), on the other hand, restores the environment saved by the most recent invocation of the Setjmp() with the corresponding **buf** argument.

```
program Setjmp

dim buf as word[ 4]
' Note: Program flow diagrams are indexed according
' to the sequence of execution

sub procedure func33()
  Delay_ms(1000)

  nop
  longjmp(buf, 2)
  nop

end sub

sub procedure func()
  PORTB = 3
  if (setjmp(buf) = 2) then
    PORTB = 1
  else
    func33()
  end if

end sub

main:
  ADPCFG = 0xFFFF

  PORTB = 0
  TRISB = 0

  nop

  func()

  nop
  Delay_ms(1000)
  PORTB = 0xFFFF
end.
```

String Library

mikroBasic for dsPIC30/33 and PIC24 includes a library which automatizes string related tasks.

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

Memchr

Prototype	`sub function memchr(dim p as word, dim ch as byte, dim n as word) as word`
Description	The function locates the first occurrence of the word ch in the initial n words of memory area starting at the address p. The function returns the offset of this occurrence from the memory address p or $FFFF if ch was not found. For the parameter p you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example @mystring or @PORTB.

Memcmp

Prototype	`sub function memcmp(dim p1, p2, n as word) as integer`
Description	The function returns a positive, negative, or zero value indicating the relationship of first n words of memory areas starting at addresses p1 and p2. This function compares two memory areas starting at addresses p1 and p2 for n words and returns a value indicating their relationship as follows: **Value** **Meaning** < 0 p1 "less than" p2 = 0 p1 "equal to" p2 > 0 p1 "greater than" p2 The value returned by the function is determined by the difference between the values of the first pair of words that differ in the strings being compared. For parameters p1 and p2 you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example @mystring or @PORTB.

Memcpy

Prototype	`sub procedure memcpy(dim p1, p2, nn as word)`
Description	The function copies nn words from the memory area starting at the address p2 to the memory area starting at p1. If these memory buffers overlap, the memcpy function cannot guarantee that words are copied before being overwritten. If these buffers do overlap, use the memmove function. For parameters p1 and p2 you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example @mystring or @PORTB.

Memmove

Prototype	`sub procedure memmove(dim p1, p2, nn as word)`
Description	The function copies nn words from the memory area starting at the address p2 to the memory area starting at p1. If these memory buffers overlap, the Memmove function ensures that the words in p2 are copied to p1 before being overwritten. For parameters p1 and p2 you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example @mystring or @PORTB.

Memset

Prototype	`sub procedure memset(dim p as word, dim character as byte, dim n as word)`
Description	The function fills the first n words in the memory area starting at the address p with the value of word character. For parameter p you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example @mystring or @PORTB.

Strcat

Prototype	`sub procedure strcat(dim byref s1, s2 as string[100])`
Description	The function appends the value of string s2 to string s1 and terminates s1 with a null character.

Strchr

Prototype	`sub function strchr(dim byref s as string[100], dim ch as byte) as word`
Description	The function searches the string s for the first occurrence of the character ch. The null character terminating s is not included in the search. The function returns the position (index) of the first character ch found in s; if no matching character was found, the function returns $FFFF.

Strcmp

Prototype	`sub function strcmp(dim byref s1, s2 as string[100]) as integer`
Description	The function lexicographically compares the contents of the strings s1 and s2 and returns a value indicating their relationship: **Value** **Meaning** < 0 s1 "less than" s2 = 0 s1 "equal to" s2 > 0 s1 "greater than" s2 The value returned by the function is determined by the difference between the values of the first pair of words that differ in the strings being compared.

Strcpy

Prototype	`sub procedure strcpy(dim byref s1, s2 as string[100])`
Description	The function copies the value of the string s2 to the string s1 and appends a null character to the end of s1.

Strcspn

Prototype	`sub function strcspn(dim byref s1, s2 as string[100]) as word`
Description	The function searches the string s1 for any of the characters in the string s2. The function returns the index of the first character located in s1 that matches any character in s2. If the first character in s1 matches a character in s2, a value of 0 is returned. If there are no matching characters in s1, the length of the string is returned (not including the terminating null character).

Strlen

Prototype	`sub function strlen(dim byref s as string[100]) as word`
Description	The function returns the length, in words, of the string s. The length does not include the null terminating character.

Strncat

Prototype	`sub procedure strncat(dim byref s1, s2 as string[100], dim size as byte)`
Description	The function appends at most size characters from the string s2 to the string s1 and terminates s1 with a null character. If s2 is shorter than the size characters, s2 is copied up to and including the null terminating character.

Strncmp

Prototype	`sub function strncmp(dim byref s1, s2 as string[100], dim len as byte) as integer`
Description	The function lexicographically compares the first len words of the strings s1 and s2 and returns a value indicating their relationship: **Value** **Meaning** `< 0` s1 "less than" s2 `= 0` s1 "equal to" s2 `> 0` s1 "greater than" s2 The value returned by the function is determined by the difference between the values of the first pair of words that differ in the strings being compared (within first len words).

Strncpy

Prototype	`sub procedure strncpy(dim byref s1, s2 as string[100], dim size as byte)`
Description	The function copies at most size characters from the string s2 to the string s1. If s2 contains fewer characters than size, s1 is padded out with null characters up to the total length of the size characters.

Strpbrk

Prototype	`sub procedure strpbrk(dim byref s1, s2 as string[100])`
Description	The function searches s1 for the first occurrence of any character from the string s2. The null terminator is not included in the search. The function returns an index of the matching character in s1. If s1 contains no characters from s2, the function returns $FFFF.

Strrchr

Prototype	`sub function strrchr(dim byref s as string[100], dim ch as byte) as word`
Description	The function searches the string s for the last occurrence of the character ch. The null character terminating s is not included in the search. The function returns an index of the last ch found in s; if no matching character was found, the function returns $FFFF.

Strspn

Prototype	`sub function strspn(dim byref s1, s2 as string[100]) as word`
Description	The function searches the string s1 for characters not found in the s2 string. The function returns the index of first character located in s1 that does not match a character in s2. If the first character in s1 does not match a character in s2, a value of 0 is returned. If all characters in s1 are found in s2, the length of s1 is returned (not including the terminating null character).

Strstr

Prototype	`sub function strstr(dim byref s1, s2 as string[100]) as word`
Description	The function locates the first occurrence of the string s2 in the string s1 (excluding the terminating null character). The function returns a number indicating the position of the first occurrence of s2 in s1; if no string was found, the function returns $FFFF. If s2 is a null string, the function returns 0.

Time Library

The Time Library contains functions and type definitions for time calculations in the UNIX time format which counts the number of seconds since the "epoch". This is very convenient for programs that work with time intervals, the difference between two UNIX time values is a real-time difference measured in seconds.

What is the epoch?
Originally it was defined as the beginning of 1970 GMT. (January 1, 1970 Julian day) GMT, Greenwich Mean Time, is a traditional term for the time zone in England.

The TimeStruct type is a structure type suitable for time and date storage.

Library Routines

```
Time_dateToEpoch
Time_epochToDate
Time_dateDiff
```

Time_dateToEpoch

Prototype	`sub function Time_dateToEpoch(dim byref ts as TimeStruct) as longint`
Returns	Number of seconds since January 1, 1970 0h00mn00s.
Description	This function returns the unix time : number of seconds since January 1, 1970 0h00mn00s. Parameters : - `ts`: time and date value for calculating unix time.
Requires	Nothing.

Example	`dim ts1 as TimeStruct`
	`  Epoch as longint`
	`...`
	`' what is the epoch of the date in ts ?`
	`epoch = Time_dateToEpoch(ts1)`

Time_epochToDate

Prototype	`sub procedure Time_epochToDate(dim e as longint, dim byref ts as TimeStruct)`
Description	Converts the unix time to time and date. Parameters : - `e`: unix time (seconds since unix epoch) - `ts`: time and date structure for storing conversion output

Example	`dim ts2 as TimeStruct`
	`  epoch as longint`
	`...`
	`' what date is epoch 1234567890 ?`
	`epoch = 1234567890`
	`Time_epochToDate(epoch,ts2)`

Time_dateDiff

Prototype	`sub function Time_dateDiff(dim byref t1, t2 as TimeStruct) as longint`
Returns	Time difference in seconds as a signed long.
Description	This function compares two dates and returns time difference in seconds as a signed long. The result is positive if `t1` is before `t2`, null if `t1` is the same as `t2` and negative if `t1` is after `t2`. Parameters : - `t1`: time and date structure (the first comparison parameter) - `t2`: time and date structure (the second comparison parameter)
Requires	Nothing.

Example	`dim ts1, ts2 as TimeStruct`
	`  diff as longint`
	`...`
	`' how many seconds between these two dates`
	`' contained in ts1 and ts2 buffers?`
	`diff = Time_dateDiff(ts1, ts2)`

Library Example

Demonstration of Time library routines usage for time calculations in UNIX time format.

```
program Time_Demo

  '*
  '* simple time structure
  '*
  structure TimeStruct
     dim ss as byte   ' seconds
     dim mn as byte   ' minutes
     dim hh as byte   ' hours
     dim md as byte   ' day in month, from 1 to 31
     dim wd as byte   ' day in week, monday=0, tuesday=1...sunday=6
     dim mo as byte
     ' month number, from 1 to 12 (and not from 0 to 11 as with
     ' unix C time !)
     dim yy as word   ' year Y2K compliant, from 1892 to 2038
  end structure

  dim  ts1, ts2     as TimeStruct
       epoch, diff as longint

  main:
    ts1.ss = 0
    ts1.mn = 7
    ts1.hh = 17
    ts1.md = 23
    ts1.mo = 5
    ts1.yy = 2006

    '*
    '* what is the epoch of the date in ts ?
    '*
    epoch = Time_dateToEpoch(ts1)       ' epoch = 1148404020


' continues...
```

```
' continued...

  '*
  '* what date is epoch 1234567890 ?
  '*

  epoch = 1234567890
  Time_epochToDate(epoch, ts2)          ' ts2.ss = 30
                                        ' ts2.mn = 31
                                        ' ts2.hh = 23
                                        ' ts2.md = 13
                                        ' ts2.wd = 4
                                        ' ts2.mo = 2
                                        ' ts2.yy = 2009

  '*
  '* how much seconds between this two dates ?
  '*
  diff = Time_dateDiff(ts1, ts2)      ' diff = 86163870

end.
```

TimeStruct type definition

```
structure TimeStruct
   dim ss as byte  ' seconds
   dim mn as byte  ' minutes
   dim hh as byte  ' hours
   dim md as byte  ' day in month, from 1 to 31
   dim wd as byte  ' day in week, monday=0, tuesday=1...sunday=6
   dim mo as byte
   ' month number, from 1 to 12 (and not from 0 to 11 as with
   ' unix C time !)
   dim yy as word  ' year Y2K compliant, from 1892 to 2038
end structure
```

Util Library

The Util library contains miscellaneous routines useful for a project development.

Library Routines

```
Lo
Hi
Higher
Highest
LoWord
HiWord
Length
```

Lo

Prototype	`sub function Lo(dim val as longint) as byte`
Returns	Low byte of `val`, bits `7..0`.
Description	The function returns low byte of `val`. The function does not interpret bit patterns of `val` – it merely returns 8 bits as found in register. Parameters : - `val`: input value
Example	`d = 0x1AC30F4` `tmp = Lo(d)  ' Equals 0xF4`

Hi

Prototype	`sub function Hi(dim val as longint) as byte`
Returns	High byte of val, bits `15..8`.
Description	The function returns high byte of `val`. The function does not interpret bit patterns of `val` – it merely returns 8 bits as found in register. Parameters : - `val`: input value
Example	`d = 0x1AC30F4` `tmp = Hib(d)  ' Equals 0x30`

Higher

Prototype	`sub function Higher(dim val as longint) as byte`
Returns	Higher byte of `val`, bits `23..16`.
Description	The function returns higher byte of `val`. The function does not interpret bit patterns of `val` – it merely returns 8 bits as found in register. Parameters : - `val`: input value
Example	`d = 0x1AC30F4` `tmp = Higherb(d)  ' Equals 0xAC`

Highest

Prototype	`sub function Highest(dim val as longint) as byte`
Returns	The highest byte of `val`, bits `31..24`.
Description	The function returns the highest byte of `val`. The function does not interpret bit patterns of `val` – it merely returns 8 bits as found in register. Parameters : - `val`: input value
Example	`d = 0x1AC30F4` `tmp = Highestb(d)  ' Equals 0x01`

LoWord

Prototype	`sub function LoWord(dim val as longint) as word`
Returns	Low word of `val`, bits `15..0`.
Description	The function returns low word of val. The function does not interpret bit patterns of `val` – it merely returns 16 bits as found in register. Parameters : - `val`: input value
Example	`a = LoWord($AC10F4)  ' Equals $10F4`

HiWord

Prototype	`sub function HiWord(dim val as longint) as word`
Returns	High word of `val`, bits `31..16`.
Description	The function returns high word of `val`. The function does not interpret bit patterns of `val` – it merely returns 16 bits as found in register. Parameters : - `val`: input value
Example	`a = HiWord($AC10F4)  ' Equals $00AC`

Length

Prototype	`sub function Length(dim byref s as string[255]) as word`
Returns	Length of the string.
Description	The function returns length of passed string. Parameters : - `s`: input string
Example	`len = Length(name)  ' len will contain length of string name`

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 the compiler

If you are experiencing any trouble with mikroBasic, 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 mikroBasic 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***
Support: ***support@mikroe.com***
Web: ***www.mikroe.com***