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

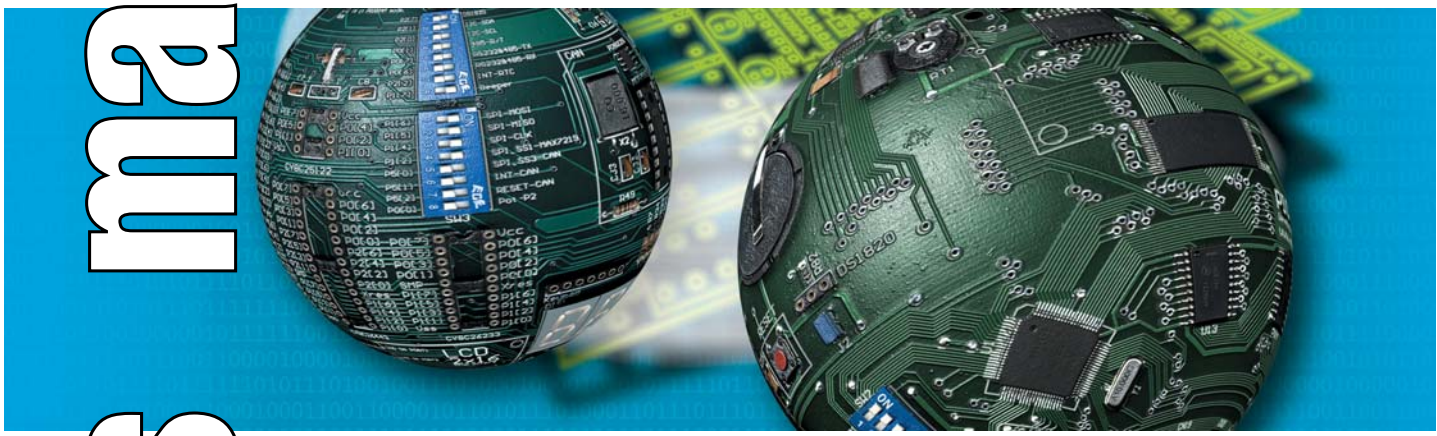
User's manual

C Compiler for Microchip dsPIC30/33 and PIC24 microcontrollers

mikroC

for dsPIC30/33 and PIC24

Making it simple



mikro ICD IN-CIRCUIT DEBUGGER

SUPPORTED from V4.0

Develop your applications quickly and easily with the world's most intuitive C compiler for dsPIC/PIC24 Microcontrollers (families dsPIC30/33 and PIC24).

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

With useful implemented tools, many practical code examples, broad set of built-in routines, and a comprehensive Help, mikroC for dsPIC30/33 and PIC24 makes a fast and reliable tool, which can satisfy needs of experienced engineers and beginners alike.

June 2007. **Reader's note**

DISCLAIMER:
mikroC for dsPIC30/33 and PIC24 and this manual are owned by mikroElektronika and are protected by copyright law and international copyright treaty. Therefore, you should treat this manual like any other copyrighted material (e.g., a book). The manual and the compiler may not be copied, partially or as a whole without the written consent from the mikroEelktronika. The PDF-edition of the manual can be printed for private or local use, but not for distribution. Modifying the manual or the compiler is strictly prohibited.

HIGH RISK ACTIVITIES
The mikroC for dsPIC30/33 and PIC24 compiler is not fault-tolerant and is not designed, manufactured or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapons systems, in which the failure of the Software could lead directly to death, personal injury, or severe physical or environmental damage ("High Risk Activities"). mikroElektronika and its suppliers specifically disclaim any express or implied warranty of fitness for High Risk Activities.

LICENSE AGREEMENT:
By using the mikroC for dsPIC30/33 and PIC24 compiler, you agree to the terms of this agreement. Only one person may use licensed version of mikroC for dsPIC30/33 and PIC24 compiler at a time.
Copyright © mikroElektronika 2003 - 2006.

This manual covers mikroC version 4.0.0.0 and the related topics. Newer versions may contain changes without prior notice.

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

- Your operating system
- Version of mikroC for dsPIC30/33 and PIC24
- 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

dsPIC, PIC24, dsPICmicro and MPLAB is a Registered trademark of Microchip company. Windows is a Registered trademark of Microsoft Corp. All other trade and/or services marks are the property of the respective owners.

mikroC for dsPIC User's manual

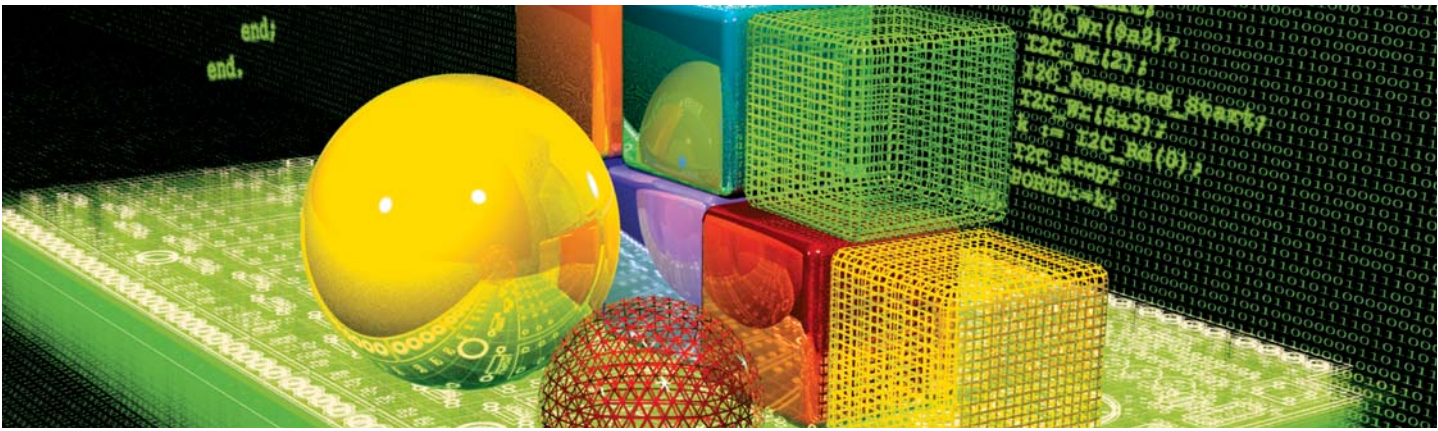


Table of Contents

CHAPTER 1 mikroC for dsPIC30/33 and PIC24 IDE

CHAPTER 2 Building Applications

CHAPTER 3 mikroC for dsPIC30/33 and PIC24 Reference

CHAPTER 4 mikroC for dsPIC30/33 and PIC24 Libraries

QUICK OVERVIEW 1
Code Editor 3
CODE EXPLORER 6
SOFTWARE SIMULATOR 7
Watch Window 8
Stopwatch Window 10
RAM Window 11
ERROR WINDOW 12
STATISTICS 13
Procedures (Graph) Window 14
Procedures (Locations) Window 14
RAM Window 15
ROM Window 15
INTEGRATED TOOLS 16
USART Terminal 16
ASCII Chart 17
7 Segment Display Decoder 18
Filter Desinger 19
UDP Terminal 20
Graphic LCD Bitmap Editor 21
KEYBOARD SHORTCUTS 22
IDE Shortcuts 22
Basic Editor shortcuts 22
Advanced Editor shortcuts 22
mikroICD Debugger and Software Simulator Shortcuts 23
PROJECTS 26
New Project 26
Edit Project 27
Extended functionality of the Project Files tab 28
Add/Remove Files from Project 28
SOURCE FILES 29
Search Paths 29
Managing Source Files 30
COMPILATION 32
Output Files 32
Assembly View 32
Error Messages 33
Compiler Warning Messages 34
mikroICD (In-Circuit Debugger) 35
mikroICD Debugger Options 36
mikroICD Debugger Example 37
mikroICD (In-Circuit Debugger) Overview 40
dsPIC30/33 and PIC24 SPECIFICS 46
Types Efficiency 46
Nested Calls Limitations 46

Limits of Indirect Approach Through PSV 46
Limits of Pointer to Function 46
mikroC SPECIFICS 47
ANSI Standard Issues 47
Predefined Globals and Constants 48
Accessing Individual Bits 48
Interrupts 49
Linker Directives 50
Code Optimization 51
Indirect Function Calls 52
LEXICAL ELEMENTS 53
Whitespace 53
Whitespace in strings 54
Line Splicing with Backslash (\) 54
Comments 55
TOKENS 57
Integer Constants 58
Floating Point Constants 60
Character Constants 61
String Constants 63
Enumeration Constants 64
Pointer Constants 64
Constant Expressions 65
KEYWORDS 66
IDENTIFIERS 67
Case Sensitivity 67
Uniqueness and Scope 67
Identifier Examples 67
PUNCTUATORS 68
Brackets 68
Parentheses 68
Braces 69
Comma 69
Semicolon 70
Colon 70
Asterisk (Pointer Declaration) 70
CONCEPTS 71
Equal Sign 71
Pound Sign (Preprocessor Directive) 71
OBJECTS AND LVALUES 72
Objects 72
Objects and Declarations 72
Lvalues 73
Rvalues 73
Scope 74

Visibility 75
NAME SPACES 76
DURATION 77
Static Duration 77
Local Duration 77
TYPES 79
Type Categories 79
FUNDAMENTAL TYPES 80
Arithmetic Types 80
Enumerations 82
Enumeration Scope 84
Void Type 84
DERIVED TYPES 85
Arrays 85
Pointers 88
Pointer Arithmetic 92
Structures 96
Unions 101
Bit Fields 102
TYPES CONVERSIONS 104
Standard Conversions 104
Explicit Types Conversions (Typecasting) 106
DECLARATIONS 108
Introduction to Declarations 108
Linkage 110
Storage Classes 112
Type Qualifiers 114
Typedef Specifier 115
asm Declaration 116
Initialization 118
FUNCTIONS 119
Function Declaration 119
Function Prototypes 120
Function Definition 121
Function Reentrancy 121
Function Calls 122
Argument Conversions 123
Ellipsis ('...') Operator 124
OPERATORS 125
Operators Precedence and Associativity 125
Arithmetic Operators 127
Relational Operators 129
Bitwise Operators 130
Logical Operators 133
Conditional Operator ? : 135

Assignment Operators 136
Sizeof Operator 138
Comma Expressions 139
STATEMENTS 141
Labeled Statements 141
Expression Statements 142
Selection Statements 142
Iteration Statements 145
Jump Statements 148
Compound Statements (Blocks) 150
PREPROCESSOR 151
Preprocessor Directives 151
Macros 152
Macros with Parameters 154
Undefining Macros 155
File Inclusion 156
Preprocessor Operators 157
Conditional Compilation 158
BUILT-IN ROUTINES 162
Hib 163
Lob 163
Higherb 164
Highestb 164
Lo 165
Hi 165
Vdelay_ms 166
Delay_ms 166
Delay_us 166
Get_Fosc_kHz 167
Delay_Cyc 167
LIBRARY ROUTINES 168
Adc_Read 169
ADC Library 169
Adc1_Read 170
Adc2_Read 171
Library Example 172
Adc_Read 172
Hardware Connection 174
Advanced SPI Ethernet Library 175
EthSetGateWayAddr 178
EthSetIPMask 178
EthSetMACAddr 178
EthSetIPAddr 178
MACIsTxReady 179
MACInit 179

EthInit....179
MACGet....180
MACGetHeader....180
MACDiscardRx....181
MACGetArray....181
MACPut....182
MACPutHeader....182
MACPutArray....183
MACFlush....183
MACSetRxBuffer....184
MACDiscardTx....184
MACSetTxBuffer....185
MACReserveTxBuffer....186
MACSetDuplex....187
MACGetFreeRxSize....187
ARPPut....188
ARPGet....188
IPIsTxReady....189
ARPInit....189
IPSetTxBuffer....190
IPPutArray....191
IPPutHeader....191
IPGetHeader....192
IPGetArray....193
IPSetRxBuffer....194
ICMPPut....195
ICMPIsTxReady....195
ICMPGet....196
TCPListen....197
TCPInit....197
TCPConnect....198
TCPIsConnected....199
TCPDisconnect....200
TCPIsPutReady....201
TCPPut....202
TCPFlush....203
TCPIsGetReady....204
TCPGet....205
TCPGetArray....206
TCPDiscard....207
TCPProcess....208
UDPOpen....209
UDPInit....209
TCPTick....209
UDPClose....210

UDPIsPutReady....211
UDPPut....212
UDPFlush....213
UDPIsGetReady....214
UDPGet....215
UDPDiscard....216
UDPProcess....217
UDPOpenSocket....218
UDPRead....218
UDPWrite....218
HTTPServer....219
HTTPInit....219
StackTask....219
StackInit....219
Library Example....220
Hardware Connection....224
Library Routines....225
CAN Library....225
CAN1GetOperationMode....226
CAN1SetOperationMode....226
CAN1Initialize....227
CAN1SetBaudRate....228
CAN1SetMask....229
CAN1SetFilter....230
CAN1Read....231
CAN1Write....232
CAN2GetOperationMode....233
CAN2SetOperationMode....233
CAN2Initialize....234
CAN2SetBaudRate....235
CAN2SetMask....236
CAN2SetFilter....237
CAN2Read....238
CAN2Write....239
CAN Constants....240
Library Example....243
Hardware Connection....245
Library Routines....246
CANSPI Library....246
CANSPI1GetOperationMode....247
CANSPI1SetOperationMode....247
CANSPI1Init....248
CANSPI1SetBaudRate....249
CANSPI1SetMask....250
CANSPI1SetFilter....251

CANSPI1Read 252
CANSPI1Write 253
CANSPI2GetOperationMode 254
CANSPI2SetOperationMode 254
CANSPI2Init 255
CANSPI2SetBaudRate 256
CANSPI2SetMask 257
CANSPI2SetFilter 258
CANSPI2Read 259
CANSPI2Write 260
CANSPI Constants 261
Library Example 264
Hardware Connection 266
Library Routines 267
Compact Flash Library 267
Cf_Init 268
Cf_Enable 269
Cf_Detect 269
Cf_Read_Byte 270
Cf_Disable 270
Cf_Read_Init 270
Cf_Write_Init 271
Cf_Read_Word 271
Cf_Write_Word 272
Cf_Write_Byte 272
Cf_Write_Sector 273
Cf_Read_Sector 273
Cf_Fat_Init 274
Cf_Fat_QuickFormat 275
Cf_Fat_Assign 276
Cf_Fat_Read 277
Cf_Fat_Reset 277
Cf_Fat_Append 278
Cf_Fat_Rewrite 278
Cf_Fat_Delete 278
Cf_Fat_Set_File_Date 279
Cf_Fat_Write 279
Cf_Fat_Get_File_Size 280
Cf_Fat_Get_File_Date 280
Cf_Fat_Get_Swap_File 281
Library Example 283
Hardware Connection 284
FIR_Radix 285
Library Routines 285
DSP (Digital Signal Processing) Library 285

IIR_Radix 286
FFT 287
VectorPower 288
Vector_Set 288
BitReverseComplex 288
VectorScale 289
Vector_Subtract 289
Vector_Multiply 290
Vector_Negate 290
Vector_Max 291
Vector_Min 291
Vector_Correlate 292
Vector_Dot 292
Vector_Add 293
Vector_Convolve 293
Matrix_Subtract 294
Matrix_Transponse 294
Matrix_Multiply 295
Matrix_Scale 295
Matrix_Add 296
Library Routines 297
ECAN Library (Enhanced Controller Area Network) 297
ECAN1DmaChannelInit 298
ECAN1GetOperationMode 299
ECAN1SetOperationMode 299
ECAN1Initialize 300
ECAN1FilterDisable 301
ECAN1SelectTxBuffers 301
ECAN1SetBufferSize 302
ECAN1FilterEnable 302
ECAN1SetBaudRate 303
ECAN1SetMask 304
ECAN1SetFilter 305
ECAN1Read 306
ECAN1Write 307
ECAN2DmaChannelInit 308
ECAN2GetOperationMode 309
ECAN2SetOperationMode 309
ECAN2Initialize 310
ECAN2FilterDisable 311
ECAN2SelectTxBuffers 311
ECAN2SetBufferSize 312
ECAN2FilterEnable 312
ECAN2SetBaudRate 313
ECAN2SetMask 314

ECAN2SetFilter 315
ECAN2Read 316
ECAN2Write 317
ECAN Constants 318
Library Example 322
Hardware Connection 324
Eeprom_Erase 325
Library Routines 325
EEPROM Library 325
Eeprom_Read 326
Eeprom_Erase_Block 326
Eeprom_Write 327
Eeprom_Write_Block 327
Library Example 328
dsPIC30: 329
Flash Memory Library 329
Library Routines 330
PIC24 and dsPIC33: 330
Flash_Write_Block 331
Flash_Erase32 331
Flash_Write_Compact 332
Flash_Write_Init 333
Flash_Write_Loadlatch4 334
Flash_Write_Loadlatch4_Compact 335
Flash_Read4 336
Flash_Write_DoWrite 336
Flash_Erase 337
Flash_Read4_Compact 337
Flash_Write 338
Flash_Write_Compact 339
Flash_Read_Compact 340
Flash_Read 340
Library Example 341
Library Routines 344
Graphic LCD Library 344
Glcd_Init 345
Glcd_Config 346
Glcd_Set_Side 347
Glcd_Set_X 347
Glcd_Read_Data 348
Glcd_Set_Page 348
Glcd_Write_Data 349
Glcd_Fill 349
Glcd_Line 350
Glcd_Dot 350

Glcd_H_Line 351
Glcd_V_Line 351
Glcd_Rectangle 352
Glcd_Box 352
Glcd_Circle 353
Glcd_Set_Font 353
Glcd_Write_Char 354
Glcd_Write_Text 354
Library Example 355
Glcd_Image 355
Hardware Connection 357
Library Routines 358
I2C Library 358
I2C_Start 359
I2C_Init 359
I2C_Wait_For_Idle 360
I2C_Restart 360
I2C_Write 361
I2C_Read 361
I2c_Set_Active 362
I2C_Stop 362
I2C1_Start 363
I2C1_Init 363
I2C1_Wait_For_Idle 364
I2C1_Restart 364
I2C1_Write 365
I2C1_Read 365
I2C2_Init 366
I2C1_Stop 366
I2C2_Restart 367
I2C2_Start 367
I2C2_Wait_For_Idle 368
I2C2_Read 368
I2C2_Stop 369
I2C2_Write 369
Library Example 370
Hardware Connection 371
Keypad_Key_Press 372
Keypad_Init 372
Library Routines 372
Keypad Library 372
Library Example 373
Keypad_Key_Click 373
Hardware Connection 375
Lcd_Custom_Config 376

Library Routines 376
LCD Custom Library (4-bit interface)....376
Lcd_Custom_Out_Cp 377
Lcd_Custom_Out....377
Lcd_Custom_Cmd....378
Lcd_Custom_Chr_Cp 378
Lcd_Custom_Chr....378
LCD Commands 379
Library Example....380
Hardware Connection 381
Lcd8_Custom_Config 382
Library Routines 382
LCD8 Custom Library (8-bit interface)....382
Lcd8_Custom_Config_TwoDataPorts 383
Lcd8_Custom_Out_CP 384
Lcd8_Custom_Out....384
Lcd8_Custom_Chr....384
Lcd8_Custom_Cmd....385
Lcd8_Custom_Chr_CP 385
Library Example....385
LCD Commands 386
Hardware Connection 387
Library Routines 388
Manchester Code Library 388
Man_Receive....389
Man_Receive_Init 389
Man_Receive_Config 389
Man_Send 390
Man_Send_Init....390
Man_Send_Config....390
Library Example....391
Man_Synchro....391
Hardware Connection 393
Multi Media Card Library....394
Library Routines 395
Mmc_Init 395
Mmc_Write_Sector 396
Mmc_Read_Sector 396
Mmc_Read_Csd 397
Mmc_Read_Cid....397
Mmc_Fat_Init....398
Mmc_Fat_QuickFormat....399
Mmc_Fat_Assign....400
Mmc_Fat_Rewrite 401
Mmc_Fat_Reset 401

Mmc_Fat_Read 401
Mmc_Fat_Delete 402
Mmc_Fat_Append 402
Mmc_Fat_Write 402
Mmc_Set_File_Date 403
Mmc_Fat_Get_File_Date 403
Mmc_Fat_Get_File_Size 404
Mmc_Fat_Get_Swap_File 405
Library Example 407
Hardware Connection 412
Library Routines 413
OneWire Library 413
Ow_Write 414
Ow_Read 414
Ow_Reset 414
Library Example 415
Hardware Connection 417
Library Routines 418
Port Expander Library 418
Expander_Read_Byte 419
Expander_Init 419
Expander_Read_PortA 420
Expander_Write_Byte 420
Expander_Read_PortAB 421
Expander_Read_PortB 421
Expander_Write_PortB 422
Expander_Write_PortA 422
Expander_Set_DirectionPortA 423
Expander_Write_PortAB 423
Expander_Set_DirectionPortAB 424
Expander_Set_DirectionPortB 424
Expander_Set_PullUpsPortB 425
Expander_Set_PullUpsPortA 425
Expander_Set_PullUpsPortAB 426
Library Example 426
Hardware Connection 427
Ps2_Init 428
Library Routines 428
PS/2 Library 428
Ps2_Key_Read 429
Ps2_Config 429
Special Function Keys 430
Library Example 431
Hardware Connection 432
Pwm_Init 433

Library Routines 433
PWM Library 433
Pwm_Stop 434
Pwm_Start 434
Pwm_Set_Duty 434
Library Example 435
Hardware Connection 435
Library Routines 436
PWM Motor Library 436
Pwm_Mc_Init 437
Pwm_Mc_Stop 438
Pwm_Mc_Start 438
Pwm_Mc_Set_Duty 438
Library Example 439
Hardware Connection 439
Library Routines 440
RS-485 Library 440
RS485Master_Receive 441
RS485Master_Init 441
RS485Slave_Init 442
RS485Master_Send 442
RS485Slave_Send 443
RS485Slave_Receive 443
Library Example 444
Hardware Connection 446
Soft_I2C_Init 447
Library Routines 447
Software I2C Library 447
Soft_I2C_Read 448
Soft_I2C_Start 448
Soft_I2C_Stop 449
Soft_I2C_Write 449
Library Example 450
Soft_Spi_Config 451
Library Routines 451
Software SPI Library 451
Soft_Spi_Read 452
Soft_Spi_Read 452
Library Example 453
Soft_Spi_Write 453
Soft_Uart_Init 455
Library Routines 455
Software UART Library 455
Soft_Uart_Write 456
Soft_Uart_Read 456

Library Example....457
Sound_Play....458
Sound_Init....458
Library Routines....458
Library Example....459
Hardware Connection....460
Spi1_Init....461
Library Routines....461
SPI Library....461
Spi1_Init_Advanced....462
Spi1_Write....464
Spi1_Read....464
Spi2_Init....465
Spi2_Init_Advanced....466
Spi2_Write....468
Spi2_Read....468
Spi_Init....469
Spi_Init_Advanced....470
Spi_Write....472
Spi_Read....472
Spi_Set_Active....473
Library Example....474
Hardware Connection....475
Library Routines....476
SPI Ethernet Library....476
SPI_Ethernet_doPacket....477
SPI_Ethernet_Init....477
SPI_Ethernet_getByte....478
SPI_Ethernet_putByte....478
SPI_Ethernet_UserTCP....479
SPI_Ethernet_UserUDP....480
HW Connection....488
Library Routines....489
SPI Graphic LCD Library....489
Spi_Glcd_Init....490
Spi_Glcd_Config....490
Spi_Glcd_Set_Page....491
Spi_Glcd_Set_Side....491
Spi_Glcd_Write_Data....492
Spi_Glcd_Read_Data....492
Spi_Glcd_Set_X....492
Spi_Glcd_Dot....493
Spi_Glcd_Fill....493
Spi_Glcd_V_Line....494
Spi_Glcd_Line....494

Spi_Glcd_Rectangle 495
Spi_Glcd_H_Line 495
Spi_Glcd_Circle 496
Spi_Glcd_Box 496
Spi_Glcd_Set_Font 497
Spi_Glcd_Write_Char 497
Spi_Glcd_Image 498
Spi_Glcd_Write_Text 498
Library Example 499
Hardware Connection 501
Spi_Lcd_Config 502
Library Routines 502
SPI LCD Library (4-bit interface) 502
Spi_Lcd_Out_Cp 503
Spi_Lcd_Out 503
Spi_Lcd_Init 503
Spi_Lcd_Cmd 504
Spi_Lcd_Chr_Cp 504
Spi_Lcd_Chr 504
LCD Commands 505
Library Example (default pin settings) 505
Hardware Connection 506
Library Routines 507
SPI LCD8 (8-bit interface) Library 507
Spi_Lcd8_Init 508
Spi_Lcd8_Config 508
Spi_Lcd8_Out_Cp 509
Spi_Lcd8_Out 509
Spi_Lcd8_Cmd 510
Spi_Lcd8_Chr_Cp 510
Spi_Lcd8_Chr 510
LCD Commands 511
Library Example (default pin settings) 511
Hardware Connection 512
SPI T6963C Graphic LCD Library 513
Library Routines 514
Spi_T6963C_Init 515
Spi_T6963C_writeCommand 516
Spi_T6963C_writeData 516
Spi_T6963C_fill 517
Spi_T6963C_waitReady 517
Spi_T6963C_setPtr 517
Spi_T6963C_dot 518
Spi_T6963C_write_char 518
Spi_T6963C_write_text 519

Spi_T6963C_rectangle 520
Spi_T6963C_line 520
Spi_T6963C_circle 521
Spi_T6963C_box 521
Spi_T6963C_sprite 522
Spi_T6963C_image 522
Spi_T6963C_setBit 523
Spi_T6963C_clearBit 523
Spi_T6963C_set_cursor 523
Spi_T6963C_displayTxtPanel 524
Spi_T6963C_displayGrPanel 524
Spi_T6963C_negBit 524
Spi_T6963C_panelFill 525
Spi_T6963C_setTxtPanel 525
Spi_T6963C_setGrPanel 525
Spi_T6963C_cursor_height 526
Spi_T6963C_txtFill 526
Spi_T6963C_grFill 526
Spi_T6963C_cursor 527
Spi_T6963C_text 527
Spi_T6963C_graphics 527
Library Example 528
Spi_T6963C_cursor_blink 528
Hardware Connection 533
T6963C Graphic LCD Library 534
Library Routines 535
T6963C_init 536
T6963C_setPtr 538
T6963C_writeCommand 538
T6963C_writeData 538
T6963C_dot 539
T6963C_fill 539
T6963C_waitReady 539
T6963C_write_char 540
T6963C_write_text 541
T6963C_rectangle 542
T6963C_line 542
T6963C_circle 543
T6963C_box 543
T6963C_sprite 544
T6963C_image 544
T6963C_setBit 545
T6963C_clearBit 545
T6963C_set_cursor 545
T6963C_displayTxtPanel 546

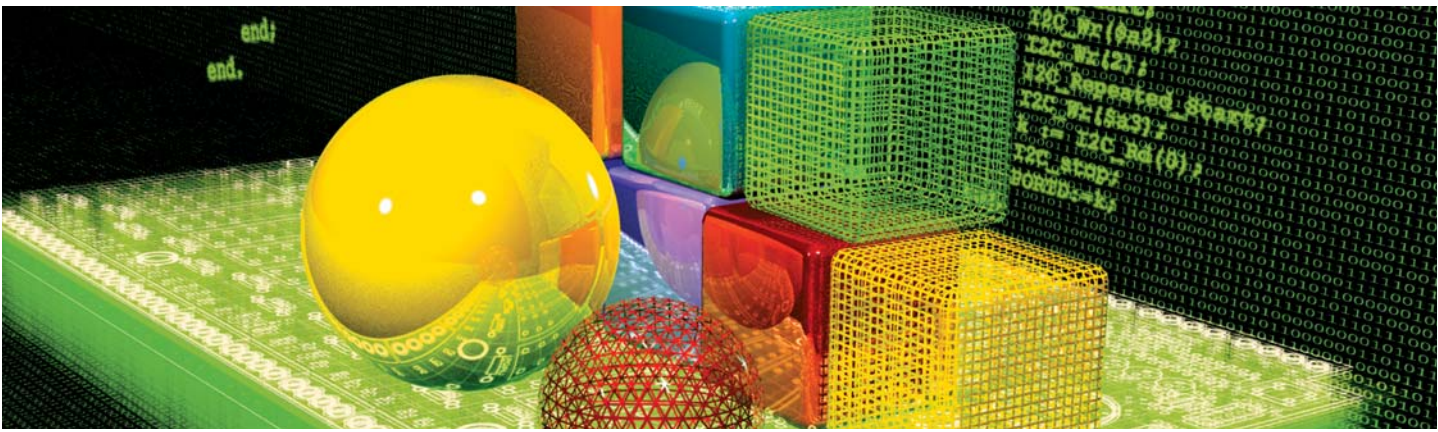
T6963C_displayGrPanel 546
T6963C_negBit 546
T6963C_panelFill 547
T6963C_setTxtPanel 547
T6963C_setGrPanel 547
T6963C_cursor_height 548
T6963C_txtFill 548
T6963C_grFill 548
T6963C_cursor 549
T6963C_text 549
T6963C_graphics 549
T6963C_cursor_blink 550
Library Example 550
Hardware Connection 555
Library Routines 556
UART Library 556
Uart1_Init 557
Uart1_Init_Advanced 558
Uart1_Data_Ready 559
Uart2_Write_Char 560
Uart1_Read_Char 560
Uart2_Init 561
Uart2_Init_Advanced 562
Uart2_Data_Ready 563
Uart2_Write_Char 564
Uart2_Read_Char 564
Uart_Init 565
Uart_Init_Advanced 566
Uart_Read_Char 567
Uart_Data_Ready 567
Uart_Set_Active 568
Uart_Write_Char 568
Uart_Set_Active 569
Hardware Connection 570
isalnum 571
Library Routines 571
ANSI C Ctype Library 571
isalpha 572
isgraph 572
isdigit 572
iscntrl 572
islower 573
isspace 573
ispunct 573
isupper 574

tolower 574
toupper 574
isxdigit 574
acos 575
Library Routines 575
ANSI C Math Library 575
ceil 576
atan2 576
atan 576
asin 576
fabs 577
exp 577
cosh 577
cos 577
log 578
ldexp 578
frexp 578
floor 578
sin 579
pow 579
modf 579
log10 579
tanh 580
tan 580
sqrt 580
sinh 580
atof 581
abs 581
Library Routines 581
ANSI C Stdlib Library 581
div 582
atol 582
atoi 582
min 583
max 583
labs 583
ldiv 583
Div Structures 584
xtoi 584
srand 584
rand 584
memchr 585
Library Routines 585
ANSI C String Library 585
memcmp 586

strcat 586
memset 586
memmove 586
memcpy 586
strlen 587
strcpy 587
strcmp 587
strchr 587
Strncmp 588
strspn 588
strncpy 588
strncat 588
Strrchr 589
Strpbrk 589
Strcspn 589
Strstr 589
ByteToStr 590
Library Routines 590
Conversions Library 590
WordToStr 591
ShortToStr 591
LongToStr 592
IntToStr 592
LongWordToStr 593
FloatToStr 593
Dec2Bcd 594
Bcd2Dec16 594
Dec2Bcd16 595
Setjmp 596
Library Routines 596
Setjmp Library 596
Longjmp 597
Library Example 598
sprintf 599
Library Routines 599
Sprint Library 599
Library Example 603
sprinti 603
sprintl 603
Time_dateToEpoch 607
Library Routines 607
Time Library 607
Time_dateDiff 608
Time_epochToDate 608
Library Example 609

SinE3 611
Library Routines 611
Trigonometry Library 611
CosE3 612
Button 613
Util Library 613
PrintOut 614
PrintOut Example 618
Contact us: 619

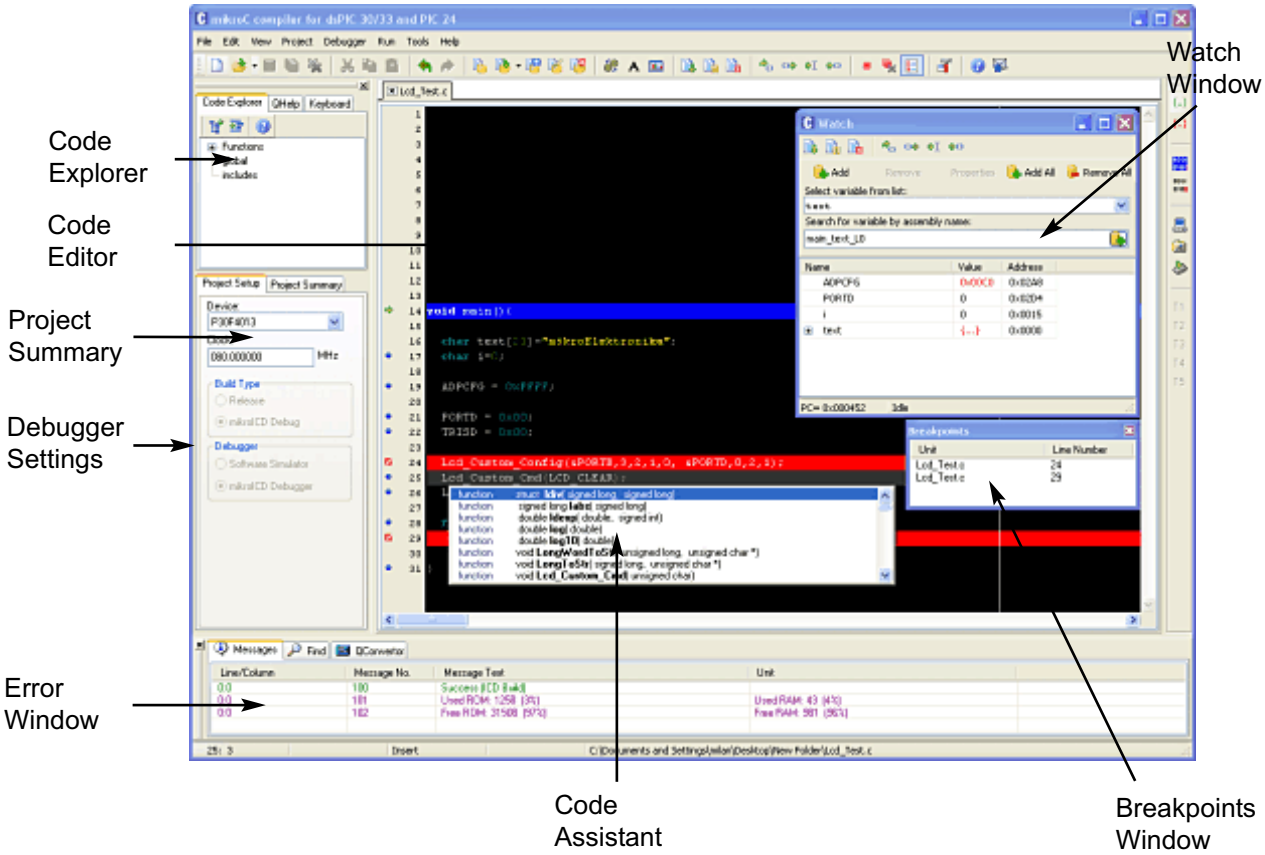
CHAPTER 1



mikroC for dsPIC30/33 and PIC24 IDE

QUICK OVERVIEW

The mikroC for dsPIC30/33 and PIC24 is a powerful, feature-rich development tool for dsPIC30/33 and PIC24 micros. It is designed to provide the programmer with the easiest possible solution to developing applications for embedded systems, without compromising performance or control. dsPIC30/33 and PIC24 and C fit together well: dsPIC is designed as PIC with digital signal processing capabilities. These are Microchip's first inherent 16-bit (data) microcontrollers. They build on the PIC's existing strength offering hardware MAC (multiply-accumulate), barrel shifting, bit reversal, (16x16)-bit multiplication and other digital signal processing operations. Having a wide range of application, being prized for its efficiency, dsPIC30/33 and PIC24 MCUs are a natural choice for developing embedded systems. mikroC provides a successful match featuring highly advanced IDE, ANSI compliant compiler, broad set of hardware libraries, comprehensive documentation, and plenty of ready-to-run examples.



mikroC allows you to quickly develop and deploy complex applications:

- Write your C source code using the built-in Code Editor (Code and Parameter Assistants, Syntax Highlighting, Auto Correct, Code Templates, and more.)
- Use included mikroC libraries to dramatically speed up the development: data acquisition, memory, displays, conversions, communication etc. Practically all dsPIC30/33 and PIC24 chips are supported.
- Monitor your program structure, variables, and functions in the Code Explorer.
- Generate commented, human-readable assembly, and standard HEX compatible with all programmers.
- Inspect program flow and debug executable logic with the integrated mikroICD Debugger or Software Simulator.
- Get detailed reports and graphs: RAM and ROM map, code statistics, assembly listing, calling tree, and more.
- dsPIC30/33 and PIC24 and C provides plenty of examples to expand, develop, and use as building bricks in your projects.

CODE EDITOR

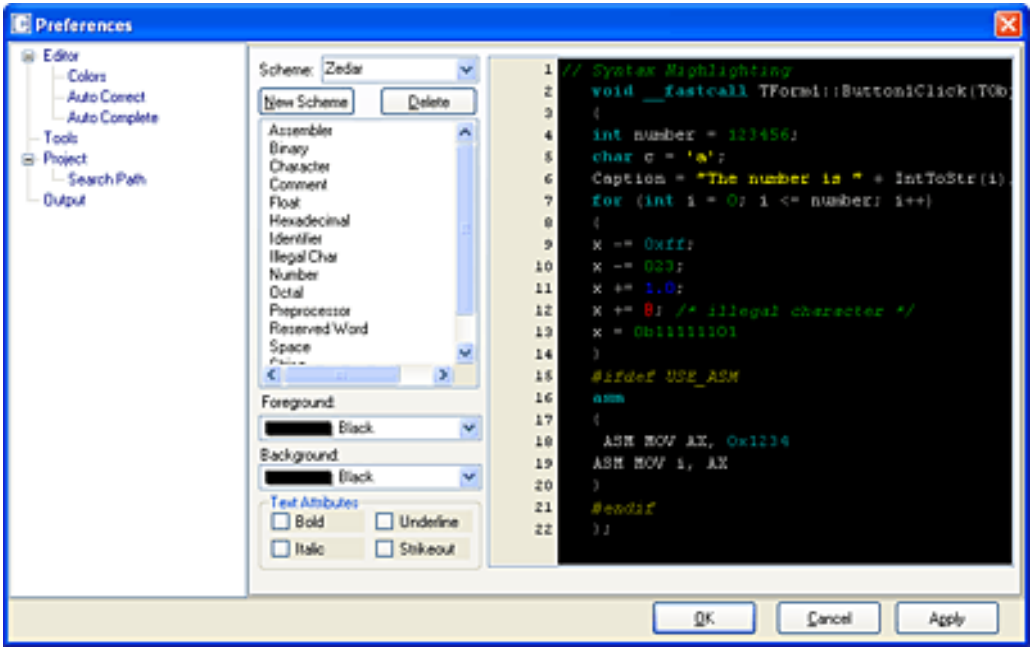
The Code Editor is advanced text editor fashioned to satisfy needs of professionals. General code editing is the same as working with any standard text-editor, including familiar Copy, Paste and Undo actions, common for Windows environment.

Advanced Editor Features:

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

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

Tools Icon.



Code Assistant [CTRL+SPACE]

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

Parameter Assistant [CTRL+SHIFT+SPACE]

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

Code Template [CTRL+J]

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

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

Auto Correct

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

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.

Comment/Uncomment

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

Bookmarks

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

Goto Line

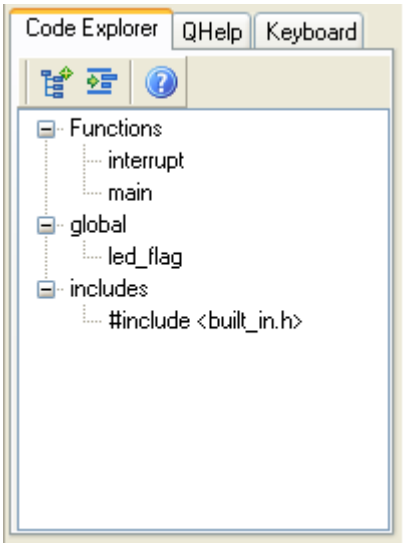
The Goto Line option makes navigation through a large code easier.
Use the shortcut Ctrl+G to activate this option.

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 right 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 the mikroC for dsPIC30/33 and PIC24.



SOFTWARE SIMULATOR

Start Debugger

The Source-level Software Simulator is an integral component of the mikroC for dsPIC30/33 and PIC24 environment. It is designed to simulate operations of the Microchip dsPIC30/33 and PIC24 MCUs and assist users in debugging C 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 (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 C (single or multi–cycle) instruction, then halt. If the instruction is a routine call, enter the routine and halt at the first instruction following the call.	[F7]
Step Over	Execute the current C (single or multi–cycle) instruction, then halt. If the instruction is a routine call, skip it and halt at the first instruction following the call.	[F8]
Step Out	Execute all remaining instructions in the current routine, return and then halt.	[Ctrl+F8]

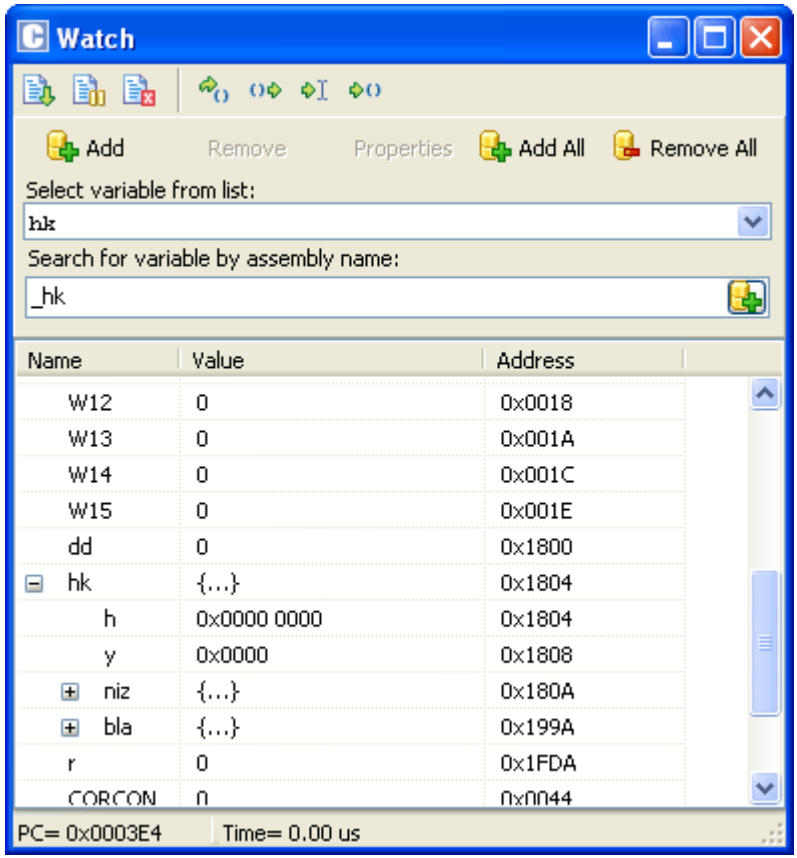
Watch Window

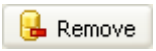
The Software Simulator Watch Window is the main Software Simulator window which allows you to monitor program items while simulating your program. To show the Watch Window, select **View › Debug Windows › Watch** from the drop-down menu.

The Watch Window displays variables and registers of the MCU, with their addresses and values. There are two ways to add variable/register into the watch list:

ADD Button

- by its real name (variable's name in "C" 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 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 will add all variables.

Remove All Button will remove all variables.

Remove All Button

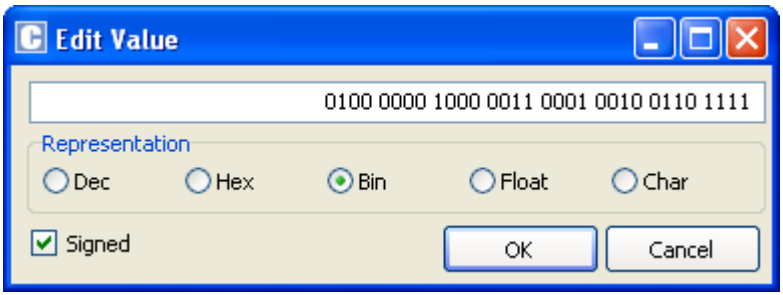
You can also expand/collapse complex variables i.e. struct type variables, strings...

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

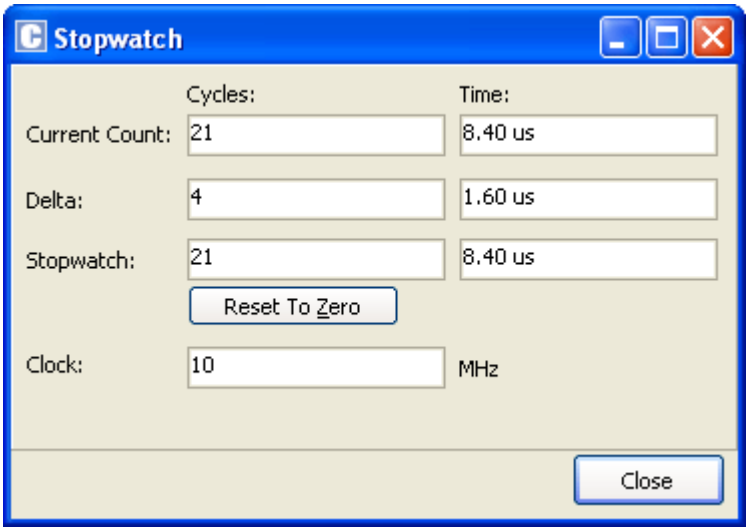


Stopwatch Window

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

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

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



RAM Window

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

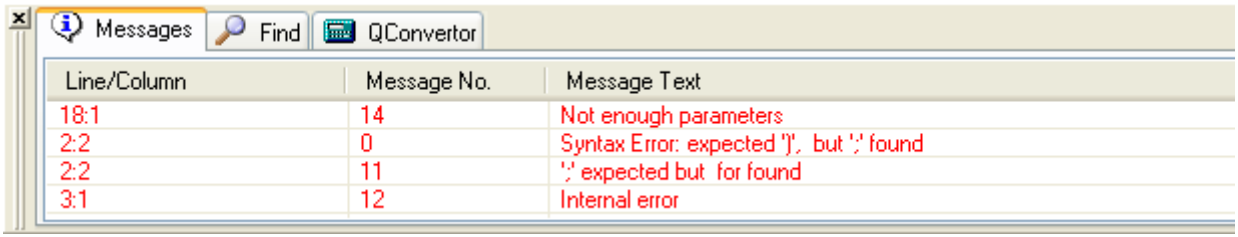
The RAM Window displays the map of MCU's RAM, with recently changed items colored red. You can change value of any field by double-clicking it.

	00	02	04	06	08	0A	0C	0E	ASCII
0000	0800	0000	0014	0999	0803	0802	0000	0000	.<BS> .. <DC4>. ™<HT> <ETX>
0010	0000	0000	0000	0000	0000	0000	0802	099A	.. .. <STX><BS> š<HT>
0020	0000	0000	0000	0000	0000	0000	0000	0446	.. .. F<EOT>
0030	0000	0000	0000	0000	0000	0000	0000	0000	.. ..
0040	0000	0000	0004	0000	0000	0000	0000	0000	.. .. <EOT>. .. ..
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	.. ..

ERROR WINDOW

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

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



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

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

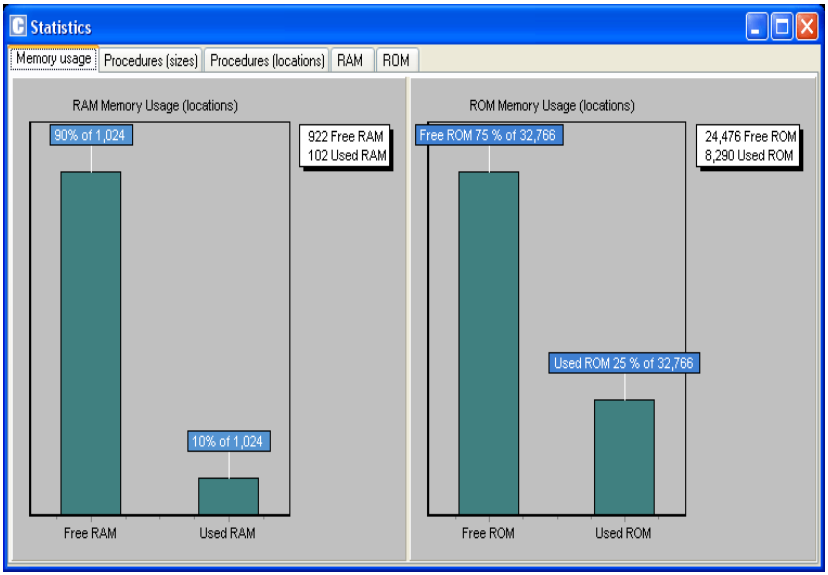
STATISTICS

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:

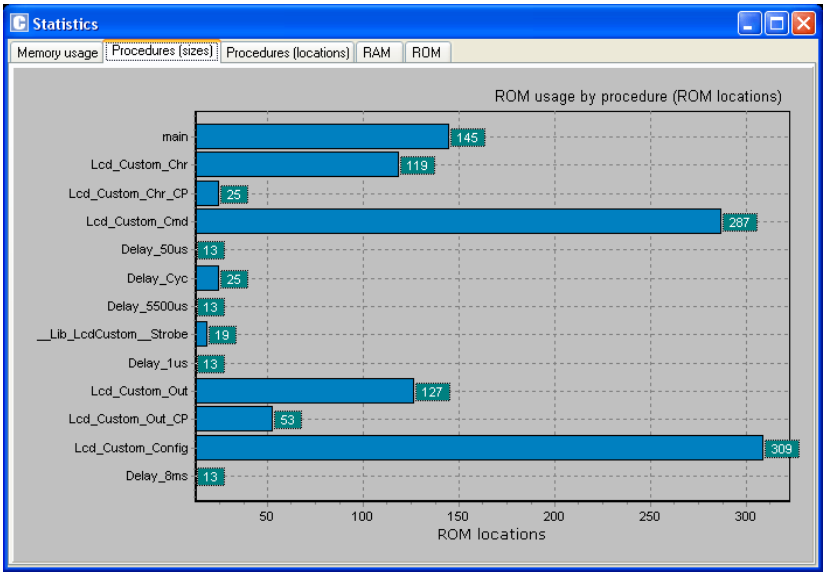
Memory Usage Window

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



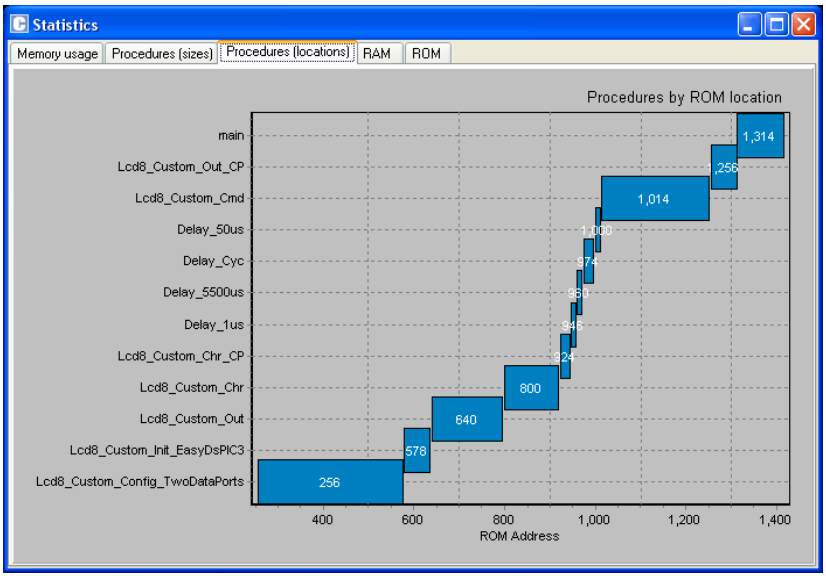
Procedures (Graph) Window

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



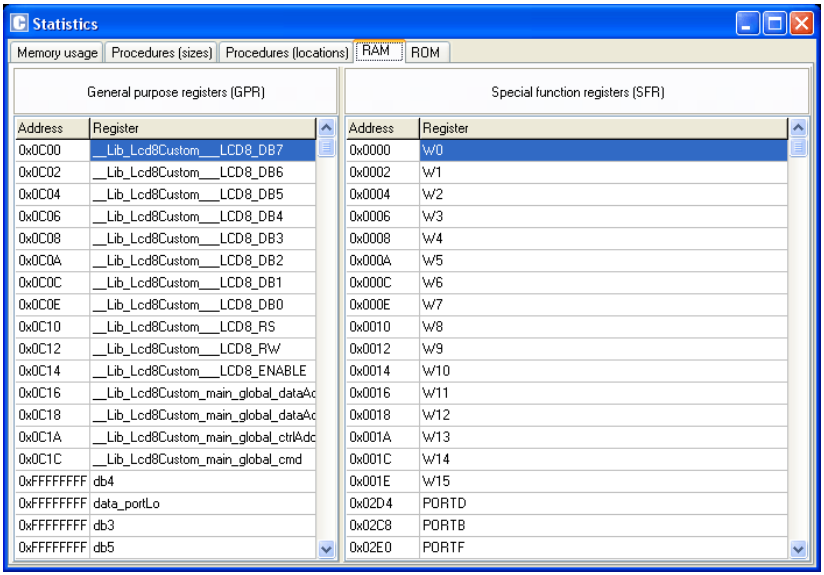
Procedures (Locations) Window

Displays how functions are distributed in microcontroller's memory.



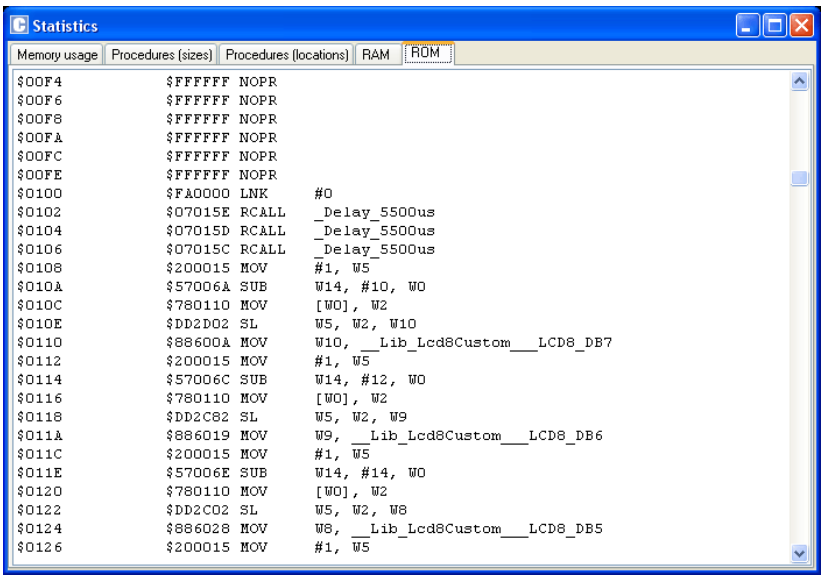
RAM Window

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



ROM Window

Lists op-codes and their addresses together with a human-readable assembler code.

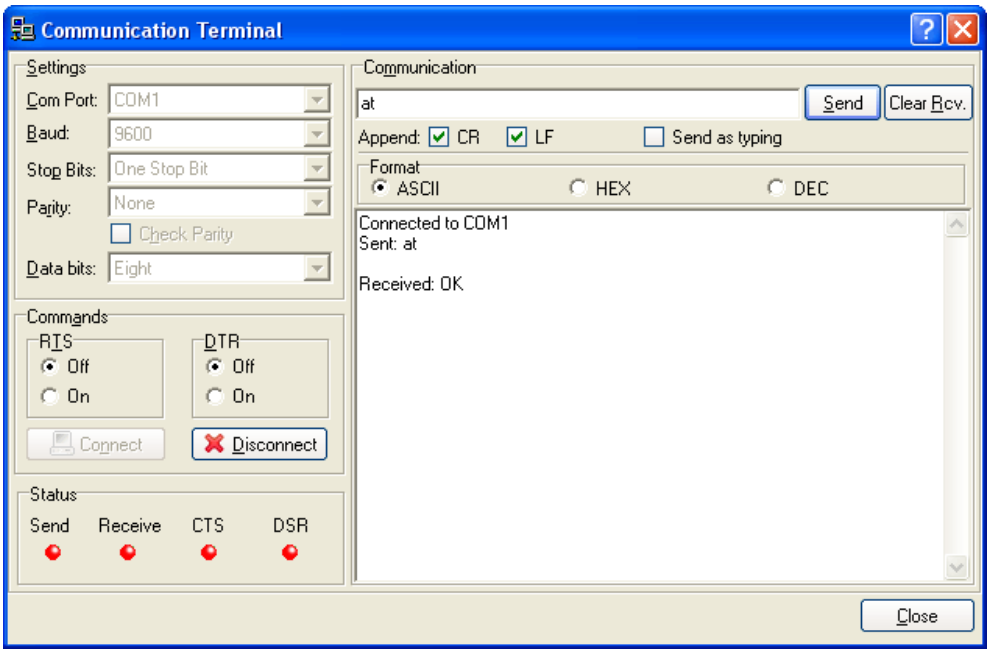


INTEGRATED TOOLS

USART Terminal Icon

USART Terminal

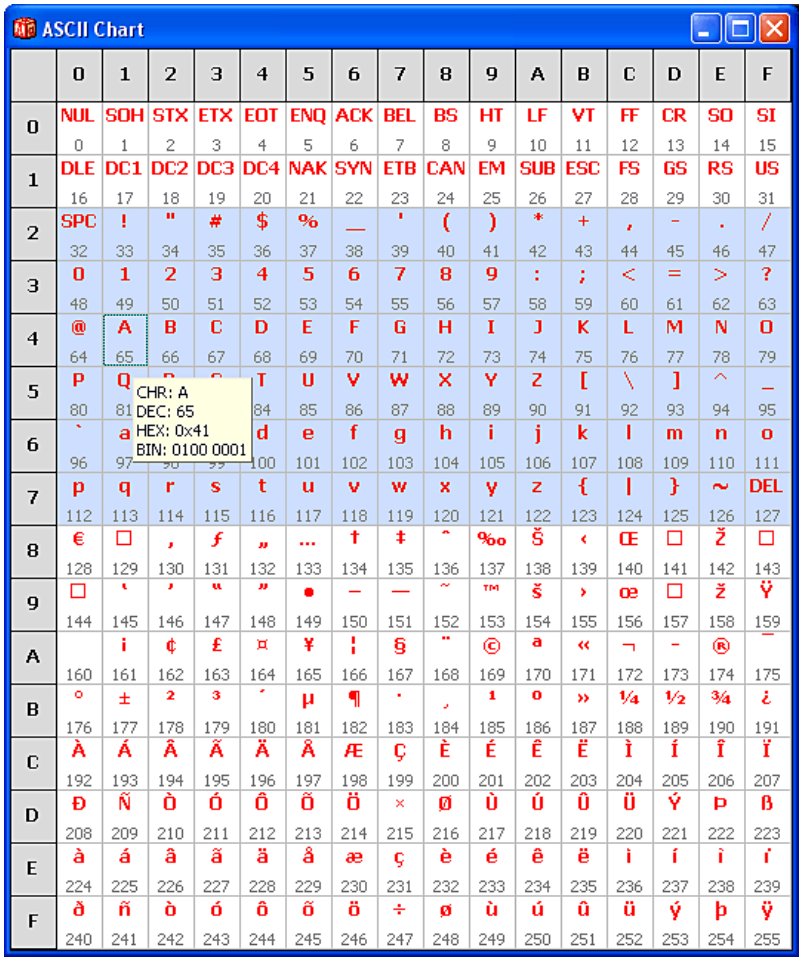
The mikroC for dsPIC30/33 and PIC24 includes the USART communication terminal for RS232 communication. You can launch it from the drop-down menu **Tools › USART Terminal** or by clicking the USART Terminal Icon .



ASCII Chart Icon

ASCII Chart

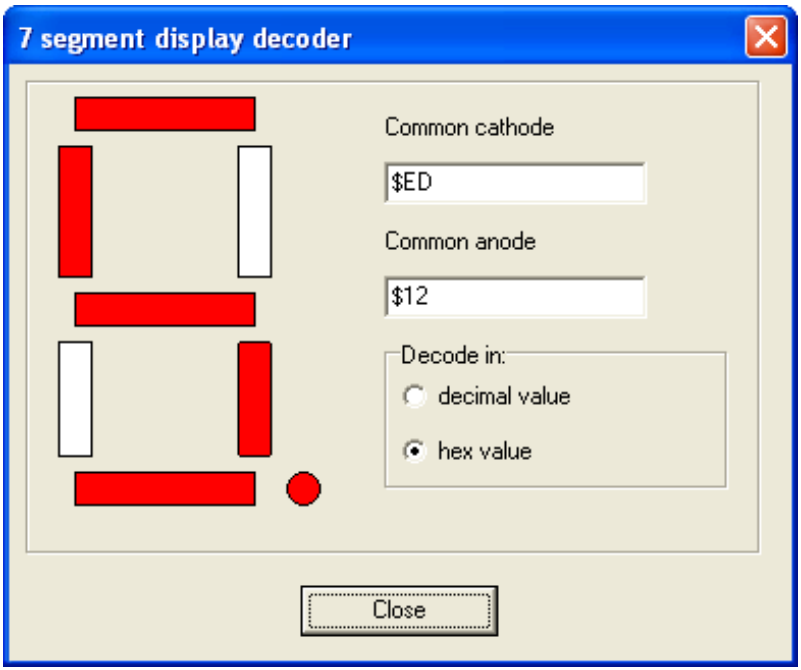
The ASCII Chart is a handy tool, particularly useful when working with LCD display. You can launch it from the drop-down menu **Tools › ASCII** chart or by clicking the View ASCII Chart Icon



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL 0	SOH 1	STX 2	ETX 3	EOT 4	ENQ 5	ACK 6	BEL 7	BS 8	HT 9	LF 10	VT 11	FF 12	CR 13	SO 14	SI 15
1	DLE 16	DC1 17	DC2 18	DC3 19	DC4 20	NAK 21	SYN 22	ETB 23	CAN 24	EM 25	SUB 26	ESC 27	FS 28	GS 29	RS 30	US 31
2	SPC 32	! 33	" 34	# 35	$ 36	% 37	_ 38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
3	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
4	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
5	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
6	` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	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

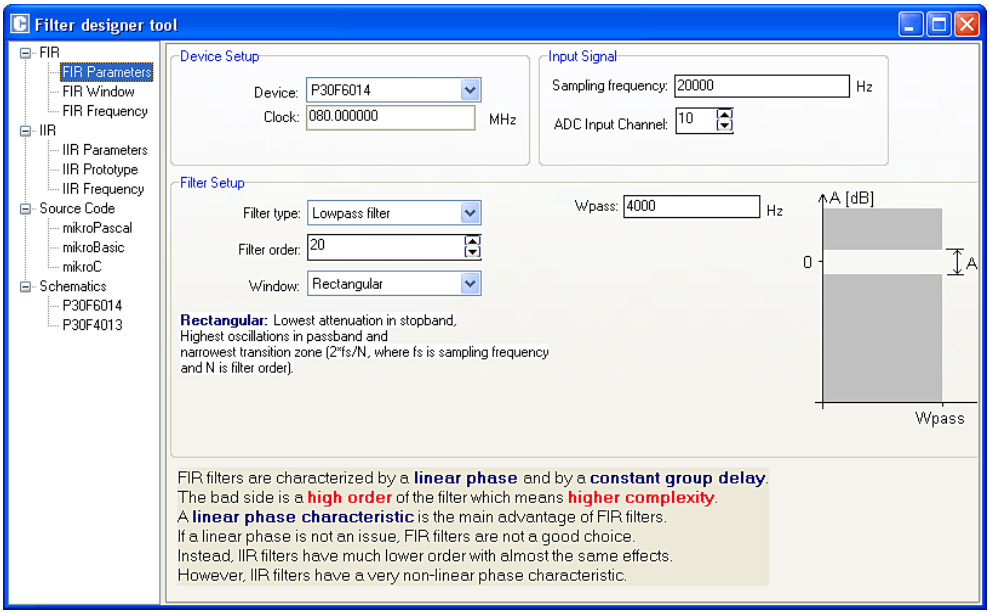
7 Segment Display Decoder

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 › 7 Segment Decoder**.



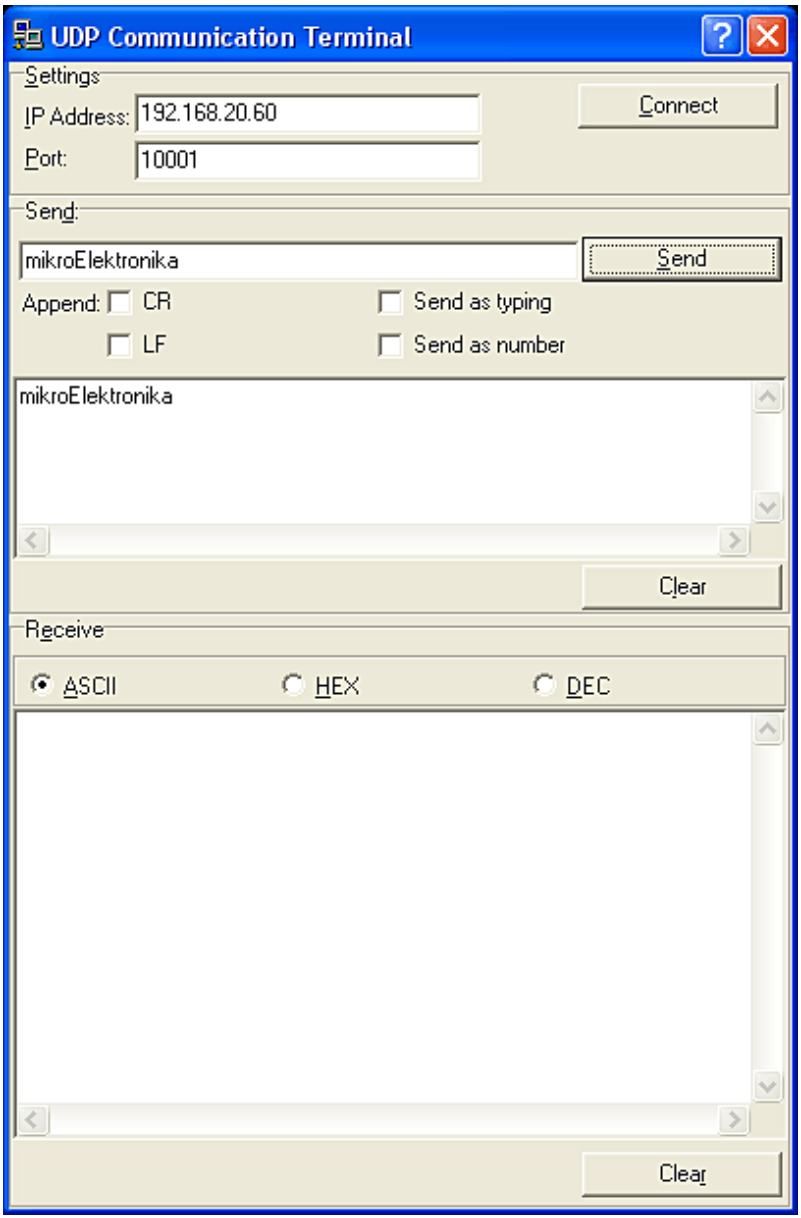
Filter Desinger

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



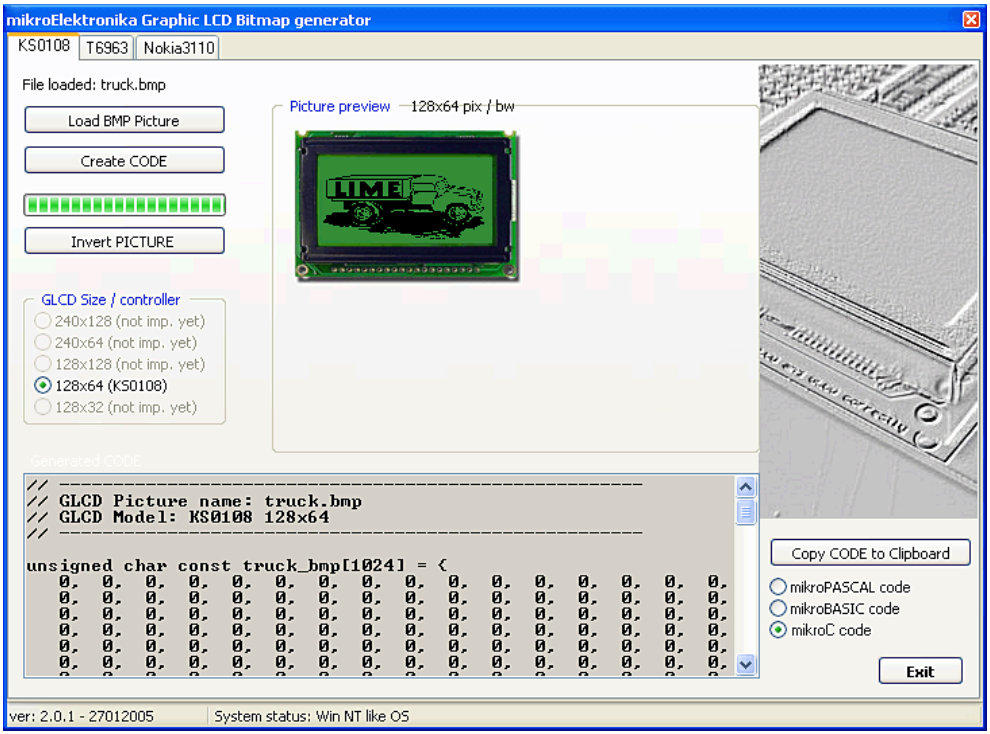
UDP Terminal

The mikroC 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

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



KEYBOARD SHORTCUTS

Below is a complete list of keyboard shortcuts available in mikroC 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
CTRL+SHIFT+F5	View breakpoints
F11	Start PICFlash Programmer

Basic Editor shortcuts

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

Advanced Editor shortcuts

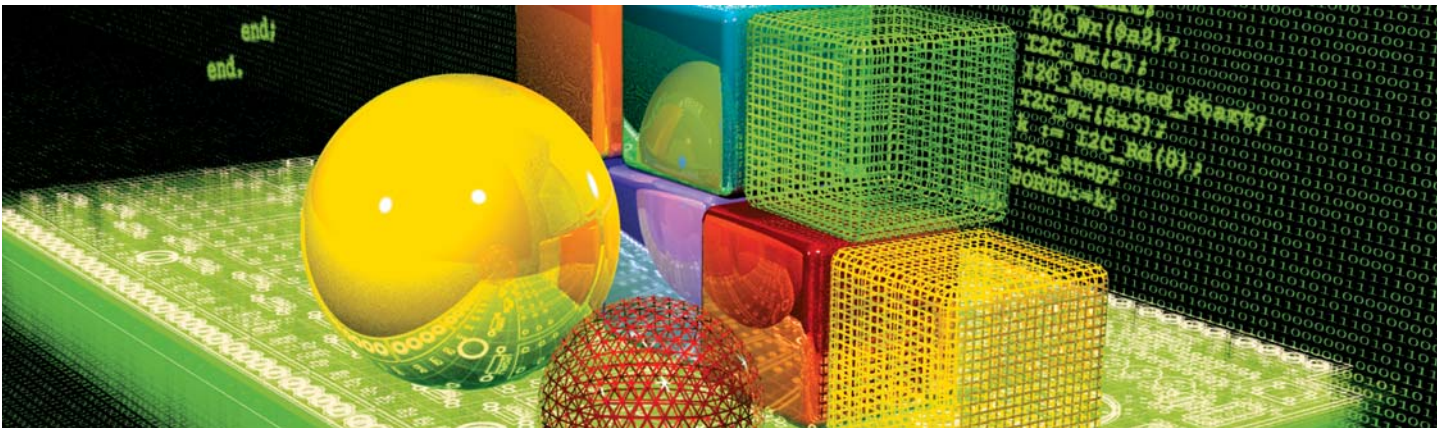
CTRL+SPACE	Code Assistant
CTRL+SHIFT+SPACE	Parameters Assistant
CTRL+D	Find declaration
CTRL+G	Goto line
CTRL+J	Insert Code Template
CTRL+<number>	Goto bookmark
CTRL+SHIFT+<number>	Set bookmark
CTRL+SHIFT+I	Indent selection
CTRL+SHIFT+U	Unindent selection

ALT+SELECT	Select columns

mikroICD Debugger and Software Simulator Shortcuts

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

CHAPTER 2



Building Applications

Creating applications in mikroC for dsPIC30/33 and PIC24 is easy and intuitive. Project Wizard allows you to set up your project in just few clicks: name your application, select chip, set flags, and get going.

mikroC for dsPIC30/33 and PIC24 allows you to distribute your projects in as many files as you find appropriate. You can then share your mikroCompiled Libraries (`.mcl` files) with other developers without disclosing the source code. The best part is that you can use `.mcl` bundles created by mikroPascal or mikroBasic!

PROJECTS

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

The project file contains the following information:

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

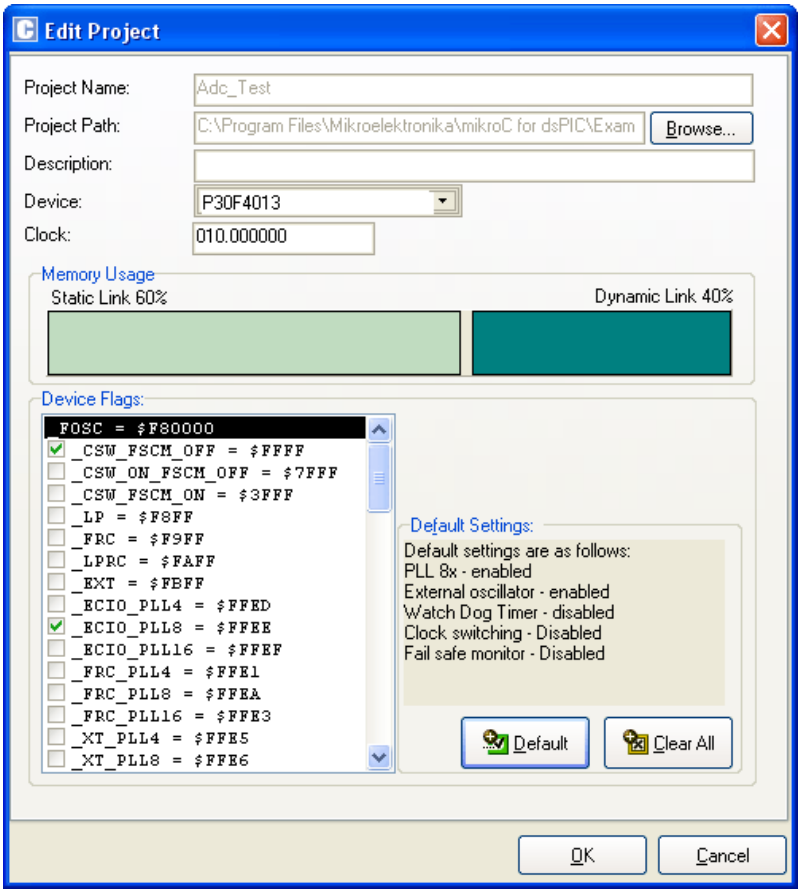
New Project

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 the mikroC for dsPIC30/33 and PIC24 will create the appropriate project file.

There is also slider for the 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 the static nor dynamic memory can occupy more than 80 % of total data memory space.

Also, an empty source file named after the project will be created by default. The mikroC for dsPIC30/33 and PIC24 does not require you to have source file named the same as the project, it's just a matter of convenience.



Edit Project.

Edit Project

Later, 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 .dpc) is stored.

Add/Remove Files from Project

The project can contain any number of source files (extension .c). The list of relevant source files is stored in the project file (extension .dpc).

Add to Project.

To add source file to the project, select **Project › Add to Project** from the drop-down menu, or click the Add to Project Icon . Each added source file must be self-contained, i.e. it must have all necessary definitions after preprocessing.

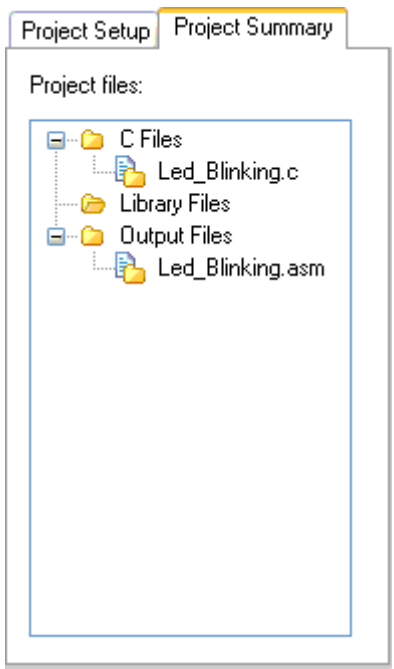
Remove from Project.

To remove file(s) from the project, select **Project › Remove from Project** from the drop-down menu, or click the Remove from the Project Icon .

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

Extended functionality of the Project Files tab

By using the Project Files' features, the user can easily reach, add or remove source library and output files (mouse right click activates the menues).



SOURCE FILES

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

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

Search Paths

Paths for Source Files (.c)

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

In the project settings, you can specify either absolute or relative path to the source file. If you specify a relative path, the mikroC for dsPIC30/33 and PIC24 will look for the file in the following locations, in this particular order:

1. the project folder (folder which contains the project file .dpc),
2. custom search paths,
3. the mikroC for dsPIC30/33 and PIC24 installation folder › “uses” folder.

Paths for Header Files (.h)

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

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

In the project settings, you can specify either absolute or relative path to the header. If you specify a relative path, the mikroC for dsPIC30/33 and PIC24 will look for the file in the following locations, in this particular order:

1. the project folder (folder which contains the project file .h),
2. the mikroC for dsPIC30/33 and PIC24 installation folder › “include” folder
3. custom search paths

New File.

Managing Source Files

Creating a new source file

To create a new source file, do the following:

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. This is a new source file.
   Select File › Save As from the drop-down menu and name it as you want.

If you use the New Project Wizard, an empty source file, named after the project with extension .c, will be created automatically. The mikroC 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.

Open File Icon.

Opening an Existing 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.

Print File Icon.

Printing an Open 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.

Save File Icon.

Saving 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

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 the dialog, browse to the folder where you want to save the file.
4. In the File Name field, modify the name of the file you want to save.
5. Click the Save button.

Closing a 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

Compile Icon.

When you have created the project and written the source code, it's time to compile it. Select **Run › Compile** from the drop-down menu, or click the Compile 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, the mikroC for dsPIC30/33 and PIC24 will generate output files.

Output Files

Upon successful compilation, the mikroC for dsPIC30/33 and PIC24 will generate output files in the project folder (folder which contains the project file .dpc). Output files are summarized in the table 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 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 the mikroC for dsPIC30/33 and PIC24, you can click the View Assembly icon or select **Project › View** Assembly from the drop-down menu to review the generated assembly code (`.asm` file) in a new tab window. Assembly is human-readable with symbolic names. All physical addresses and other information can be found in the Statistics Window or in List File (`.lst`).

ERROR MESSAGES

Error Messages

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

- Function cannot return array
- Inconsistent storage class
- Inconsistent type
- %s tag redefined
- Illegal typecast
- %s is not a valid identifier
- Invalid statement
- Constant expression required
- Internal error %s
- Too many arguments
- Not enough parameters
- Invalid expression
- Identifier expected, but %s found
- Operator [%s] not applicable to these operands [%s]
- Assigning to non-lvalue [%s]
- Cannot cast [%s] to [%s]
- Cannot assign [%s] to [%s]
- lvalue required
- Pointer required
- Argument is out of range
- Undeclared identifier [%s] in expression
- Too many initializers
- Cannot establish this baud rate at %s MHz clock

Compiler Warning Messages

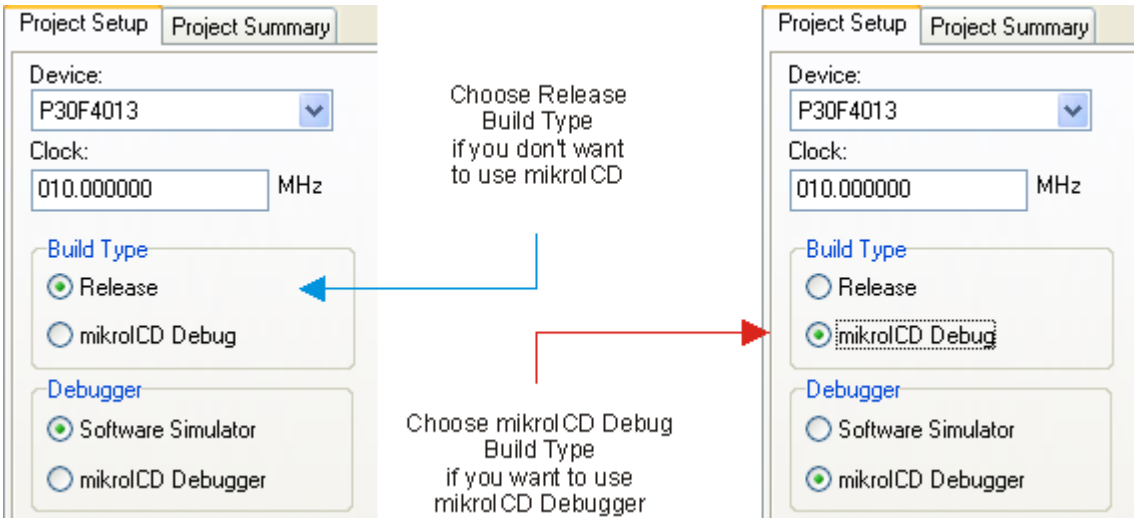
- Highly inefficent code: func call in complex expression
- Inefficent code: func call in complex expression

mikroICD (In-Circuit Debugger)

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

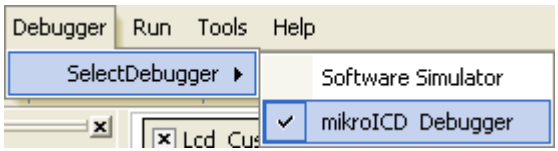
Step No. 1

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 the Debugger makes more options available: Step Into, Step Over, Run to Cursor, etc. Line that is to be executed is color highlighted (blue by default). There is also notification about the program execution and it can be found in the Watch Window (yellow status bar). Note that some functions take more time to execute; execution is indicated with "Running..." message in the Watch Window Status Bar.

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 C (single or multi–cycle) instruction, then halt. If the instruction is a routine call, enter the routine and halt at the first instruction following the call.	[F7]
Step Over	Execute the current C (single or multi–cycle) instruction, then halt. If the instruction is a routine call, skip it and halt at the first instruction following the call.	[F8]
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.

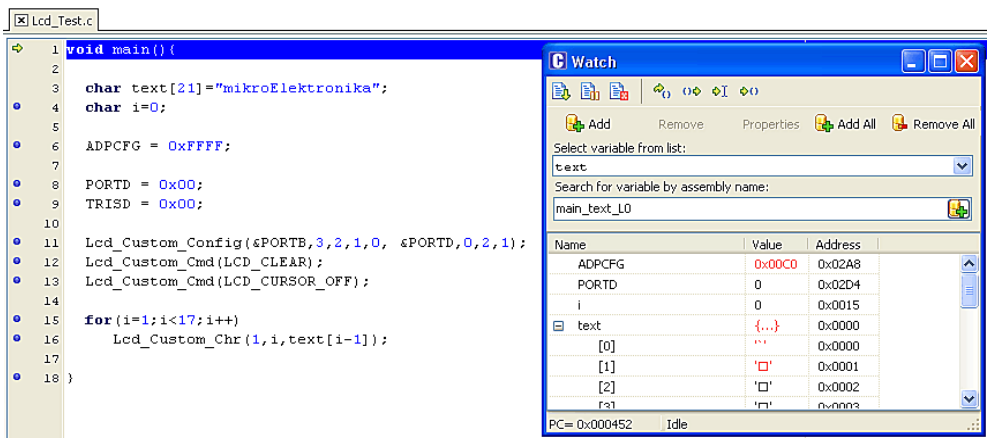
Step No. 1

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

```
void main(){

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

        ADPCFG = 0xFFFF;

        PORTD = 0x00;
        TRISD = 0x00;

        Lcd_Custom_Config(&PORTB,3,2,1,0, &PORTD,0,2,1);
        Lcd_Custom_Cmd(LCD_CLEAR);
        Lcd_Custom_Cmd(LCD_CURSOR_OFF);

        for(i=1;i<17;i++)
            Lcd_Custom_Chr(1,i,text[i-1]);

}
```

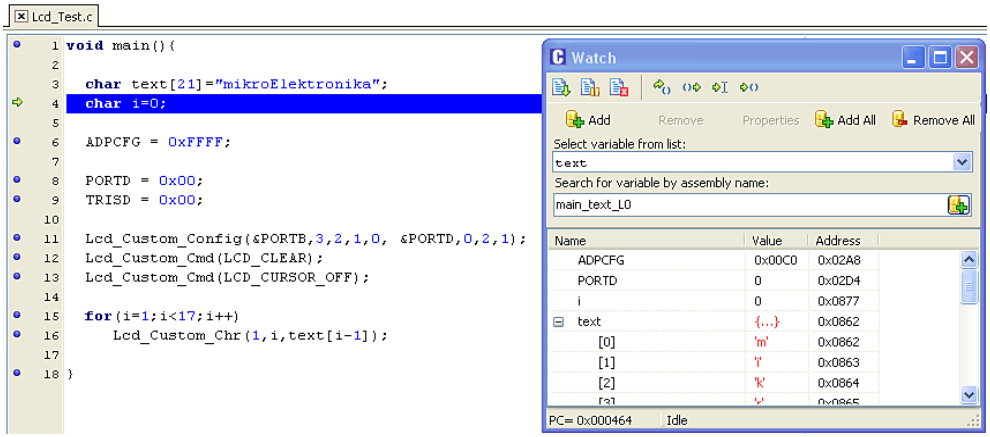
Step No. 2

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

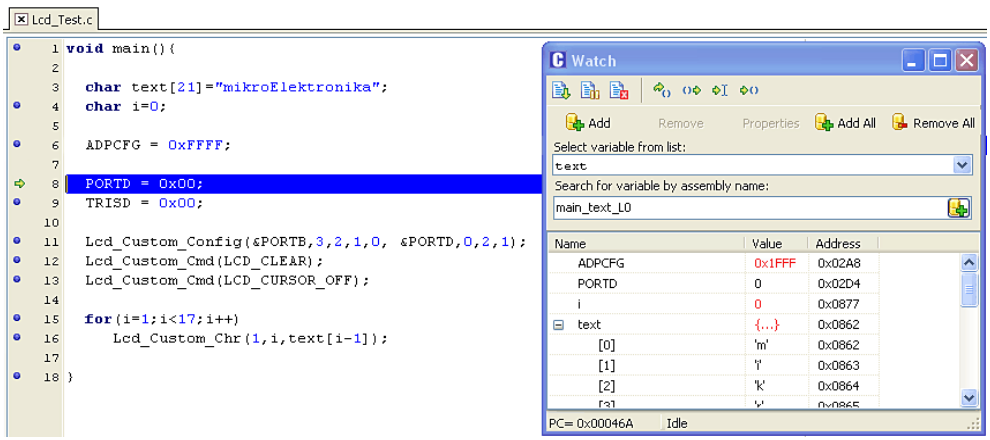


Step No. 3

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

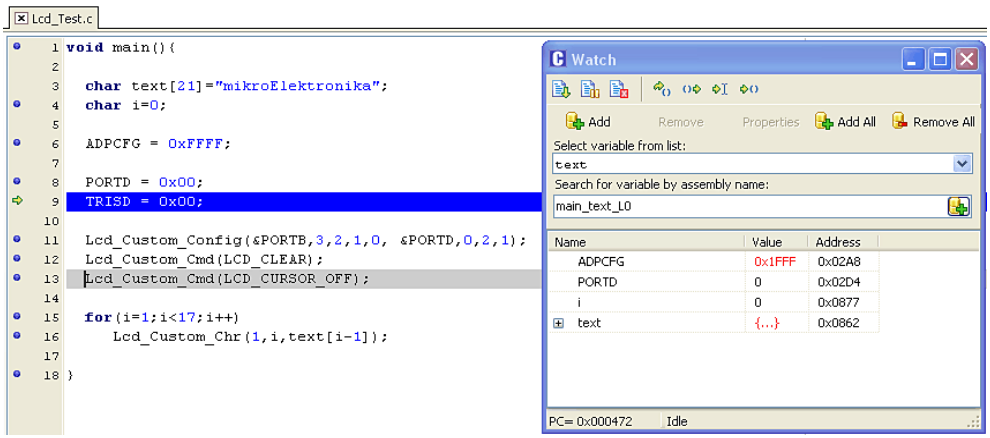


All changes are read from MCU and loaded into the Watch Window. Note that **ADPCFG** has changed its value to 0x1FFF.



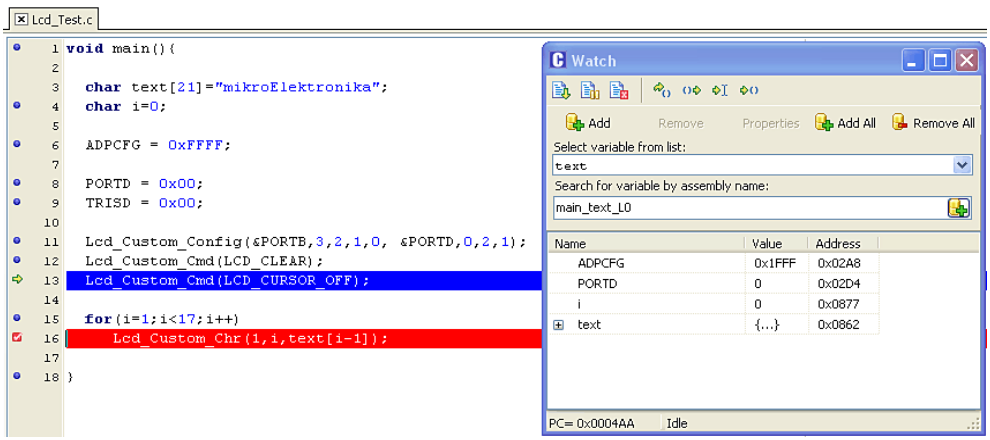
Step No. 4

Step Into [**F7**] and Step Over [**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.

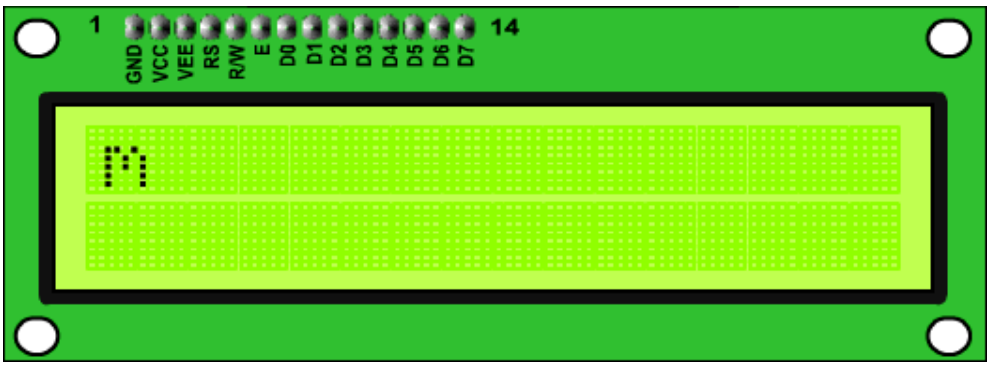


Step No. 5

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" on LCD char by char. 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.



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 (4 for PIC24 and dsPIC33 family, for dsPIC30 family this number depends on the MCU used). If all hardware brekpoints 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. The picture below demonstrates step-by-step execution of the code used in above mentioned examples.



mikroICD (In-Circuit Debugger) Overview

Watch Window

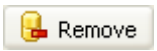
Debugger Watch Window is the main Debugger window which allows you to monitor program execution. To show the Watch Window, select **Debug Windows › Watch** from the **View** drop-down menu.

The Watch Window displays variables and registers of the MCU, 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 to add variable/register into the watch list:

ADD Button

- by its real name (variable's name in "C" 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 the variable by assemby name*** box and click the Add Button .



Remove Button

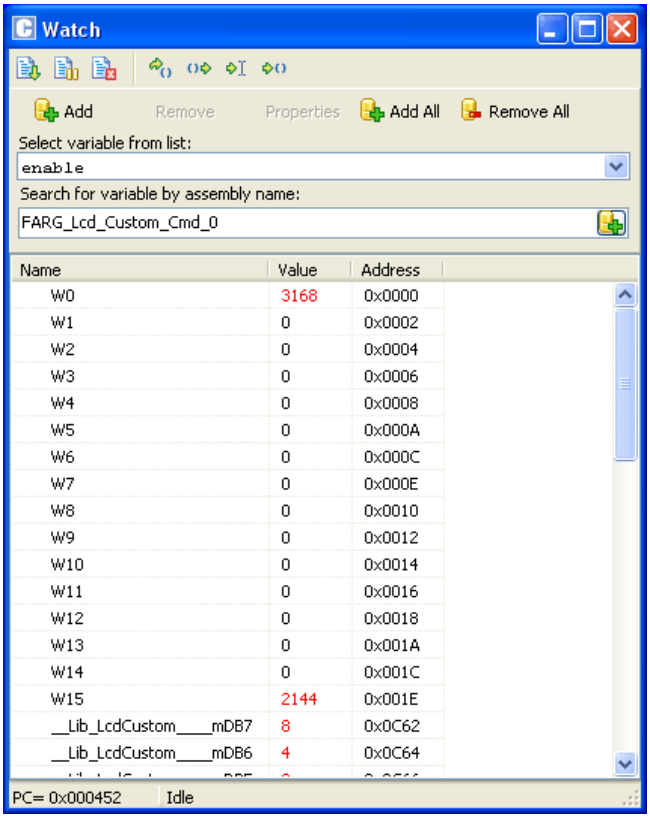
Variables can also be removed from the Watch window, just select the variable that you want to remove and then click the Remove Button .

Add All Button

Add All Button will add all variables.

Remove All Button

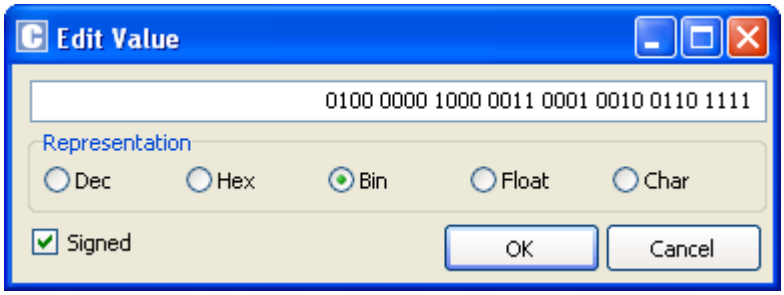
Remove All Button will remove all variables.



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.

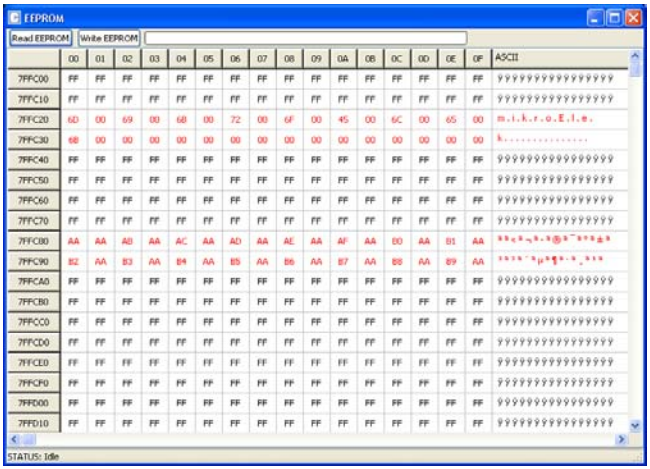


EEPROM Window

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 memory. There are two action buttons concerning the EEPROM watch window :

- **Flush EEPROM**. Writes data from the EEPROM window into MCU's internal EEPROM memory.
- **Read EEPROM**. Reads data from MCU's internal EEPROM memory and loads it up into the EEPROM window.

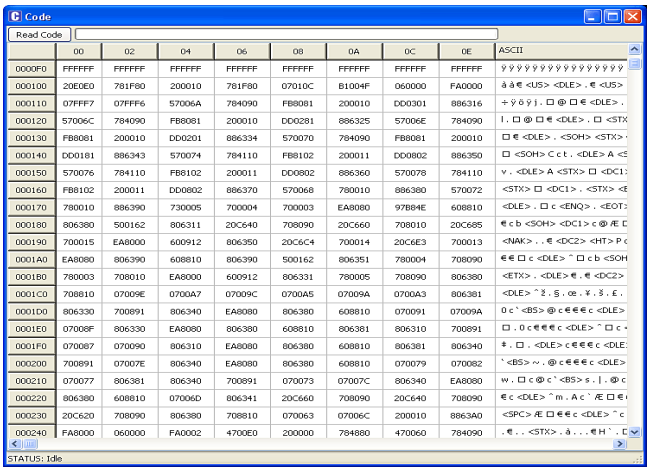


Code Window

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.

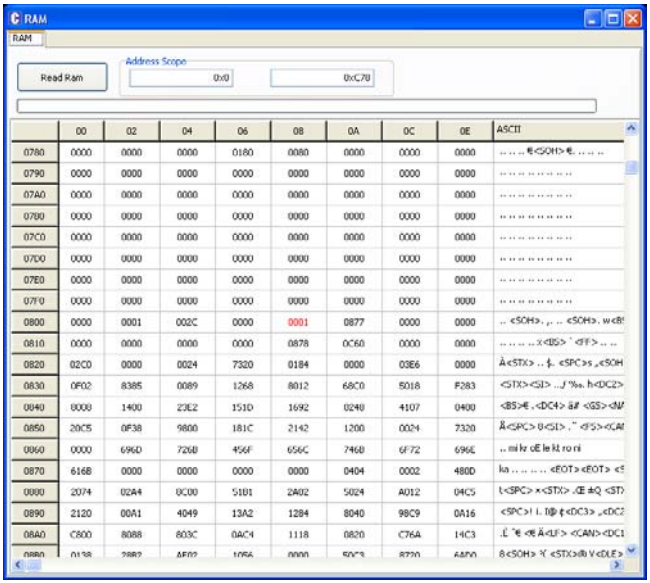


RAM Window

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.

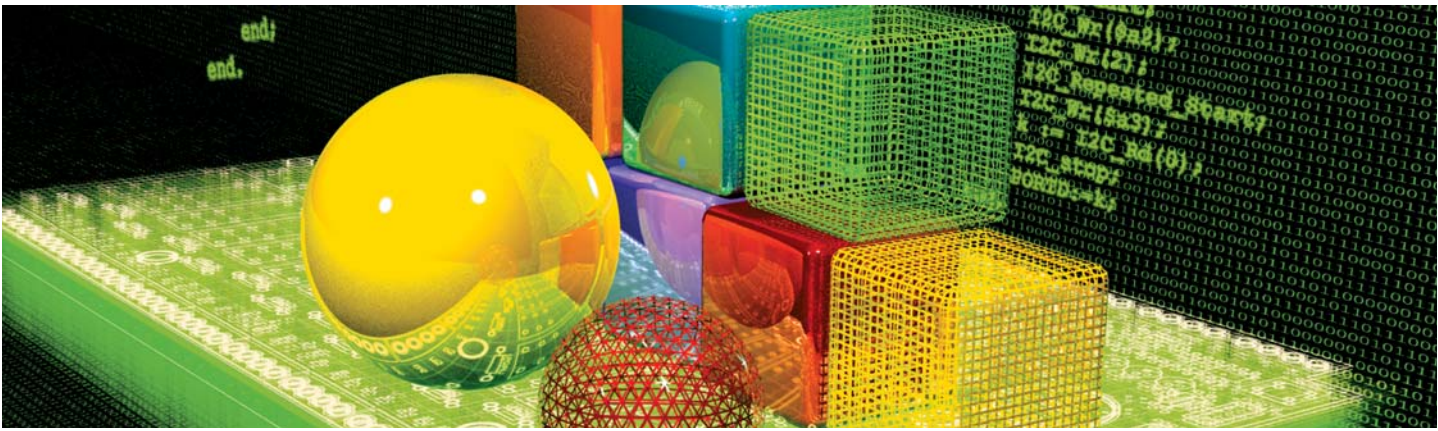
The user can edit and change the values in the RAM window. RAM window content will be written to the MCU before the next instruction execution.



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 mikroC dsPIC30/33 and PIC24 Library routines. Use **Step Over** [**F8**] command for these routines.
- Trying to use **Code Protection** or **PBOR** programmer options while using the mikroICD.
- Using mikroICD with Code Protection fuse ON.

CHAPTER 3



mikroC for dsPIC30/33 and PIC24 Language Reference

C offers unmatched power and flexibility in programming microcontrollers. mikroC for dsPIC30/33 and PIC24 adds even more power with an array of libraries, specialized for dsPIC30/33 and PIC24 HW modules and communications. This chapter should help you learn or recollect C syntax, along with the specifics of programming dsPIC30/33 and PIC24 microcontrollers. If you are experienced in C programming, you will probably want to consult mikroC for dsPIC30/33 and PIC24 Specifics first.

dsPIC30/33 and PIC24 SPECIFICS

In order to get the most from the mikroC 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 dsPIC30/33 and PIC24's capabilities and limitations, and their impact on the code writing.

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 int type.Although the mikroC for dsPIC30/33 and PIC24 is capable of handling types like char or short, dsPIC30/33 and PIC24 will generate a better code for int type so use char and short only in places where you can significantlly save RAM (e.g. for arrays char a[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 the mikroC 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 but it shouldn't be forgotten what this kind of programming technics causes.

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.

Limits of Pointer to Function

Currentlly pointer to functions are 16-bit variables. For functions which address exceeds 16 bit limit, the compiler uses handle (16-bit pointer on GOTO). A handle usage is automatic compiler process so there is no need for the user to intervene.

mikroC SPECIFICS

ANSI Standard Issues

Divergence from the ANSI C Standard

The mikroC for dsPIC30/33 and PIC24 diverges from the ANSI C standard in a few areas. Some of these modifications are improvements intended to facilitate dsPIC programming, while others are the result of dsPIC30/33 and PIC24 hardware limitations.

- Case Sensitivity. Check identifiers
- The mikroC for dsPIC30/33 and PIC24 treats identifiers declared with the const qualifier as “true constants” (C++ style). This allows using const objects in places where ANSI C expects a constant expression. If aiming at portability, use the traditional preprocessor defined constants. See Type Qualifiers and Constants.
- The mikroC for dsPIC30/33 and PIC24 allows C++ style single–line comments using two adjacent slashes (//). The comment can start at any position and extends until the next new line. See Comments.
- A number of standard C libraries (ctype, math, stdlib, string) have been imple mented; check the individual functions for divergence.
- The mikroC for dsPIC30/33 and PIC24 does not provide automatic initialization for objects. Uninitialized globals and objects with static duration will take ran dom values from memory.

Features currently under construction:
Anonymous structures and unions are not supported at present.

Implementation-defined Behavior

Certain sections of the ANSI standard have implementation-defined behavior. This means that the exact behavior of some C code can vary from compiler to compiler. This Help contains the sections describing how the mikroC for dsPIC30/33 and PIC24 compiler behaves in such situations.

The most notable specifics include:

- Storage Classes
- Bit Fields

Predefined Globals and Constants

To facilitate dsPIC30/33 and PIC24 programming, the mikroC for dsPIC30/33 and PIC24 implements a number of predefined globals and constants.

All dsPIC30/33 and PIC24 **SFR registers** are implicitly declared as global variables of volatile unsigned int. These identifiers have an external linkage, and are visible in the entire project. When creating a project, the mikroC for dsPIC30/33 and PIC24 will include an appropriate (*.c) file from defs folder, containing declarations of available **SFR registers** and constants (such as PORTB, ADPCFG, etc). All identifiers are in upper case, identical to nomenclature in the Microchip datasheets. All dsPIC30/33 and PIC24 **SFR registers** are also available as structures with bitfields named identically to the Microchip datasheets in order to facilitate bit access e.g

```
TRISBbits.TRISB3 = 1.
```

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

Accessing Individual Bits

The mikroC 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 one of identifiers F0, F1, … , F15 with F15 being the most significant bit.

There is no need for any special declarations; this kind of selective access is an intrinsic feature of the mikroC for dsPIC30/33 and PIC24 and can be used anywhere in the code. Identifiers F0–F15 are not case sensitive and have a specific namespace. You may override them with your own members F0–F15 within any given structure.

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

```
// Clear TRISB3
TRISBbits.TRISB3 = 0;
```

See Predefined Globals and Constants for more information on register/bit names.
Note: If aiming at portability, avoid this style of accessing individual bits, use the bit fields instead.

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 afterwards which connects 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.
ISR's are organized in IVT. ISR is defined as standard function but with `org 0x26` directive afterwards. `0x26` is IVT address of U1RX(UART1 Receiver) interrupt source.For more info about IVT can be found in dsPIC30/33 and PIC24 Family Reference Manual.

Function Calls from Interrupt

Calling functions from within 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 them two, saving only the registers that have been used in both threads. It is not recommended to use function call from interrupt. In case of doing that take care of stack depth.
Here is a simple example of handling the interrupts from Timer1 (if no other interrupts are allowed):

```
//-------------- Interrupt routine
void Timer1Int() org 0x1A {
  //** it is necessary to clear manually the interrupt flag:
  IFS0 = IFS0 & 0xFFF7;    // Clear TMR1IF
  //** user code starts here
  LATB = ~ PORTB;          // Invert PORTB
  //** user code ends here
} //~!
```

Linker Directives

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

Directive absolute

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

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

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

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

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

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

Directive org

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

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

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

Note: See also funcall pragma.

Code Optimization

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

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

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

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

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

"Dead code" 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.

Indirect Function Calls

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

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

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

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

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

LEXICAL ELEMENTS

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

In the tokenizing phase of compilation, the source code file is parsed (that is, broken down) into *tokens* and *whitespace*. The tokens in the mikroC for dsPIC30/33 and PIC24 are derived from a series of operations performed on your programs by the compiler and its built-in *preprocessor*.

Whitespace

Whitespace is a collective name given to spaces (blanks), horizontal and vertical tabs, newline characters and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, two sequences:

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

and

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

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

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

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

Whitespace in strings

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

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

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

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

Line Splicing with Backslash (\)

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

```
"mikroC \

Compiler"
```

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

Comments

Comments are pieces of a text used to annotate a program and technically are another form of whitespace. Comments are for the programmer's use only; they are stripped from the source text before parsing. There are two ways to delineate comments: the C method and the C++ method. Both are supported by mikroC for dsPIC30/33 and PIC24. You should also follow the guidelines on the use of white-space and delimiters in comments, discussed later in this topic to avoid other portability problems.

C comments

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

In the mikroC for dsPIC30/33 and PIC24,

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

parses as:

```
int i;
```

Note that the mikroC for dsPIC30/33 and PIC24 does not support a nonportable token pasting strategy using /**/. For more information on token pasting, refer to the Preprocessor Operators.

C++ comments

The mikroC for dsPIC30/33 and PIC24 allows single-line comments using two adjacent slashes (//). The comment can start in any position and extends until the next new line.

The following code

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

parses as:

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

Nested comments

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

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

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

```
i ; */
```

which would generate a syntax error..

TOKENS

Token is the smallest element of a C program that compiler can recognize. The parser separates tokens from the input stream by creating the longest token possible using the input characters in a left–to–right scan.
The mikroC for dsPIC30/33 and PIC24 recognizes the following kinds of tokens:

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

Tokens can be concatenated (pasted) by means of the preprocessor operator ##. See the Preprocessor Operators for details.

Token Extraction Example

Here is an example of token extraction. Take a look at the following example code sequence:

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

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

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

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

Note that +++ parses as ++ (the longest token possible) followed by +.
According to the operator precedence rules, our code sequence is actually:

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

CONSTANTS

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

The mikroC for dsPIC30/33 and PIC24 supports:

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

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

Integer Constants

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

Long and Unsigned Suffixes

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

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

Otherwise:

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

Decimal

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

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

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

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

Hexadecimal

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

Binary

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

Octal

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

Floating Point Constants

A floating-point constant consists of:

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

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

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

The mikroC for dsPIC30/33 and PIC24 limits floating-point constants to the range `±1.17549435082 * 10-38 .. ±6.80564774407 * 1038.`

Here are some examples:

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

The mikroC for dsPIC30/33 and PIC24 floating-point constants are of the type double. Note that the mikroC for dsPIC's implementation of ANSI Standard considers float and double (together with the long double variant) to be the same type.

Character Constants

A character constant is one or more characters enclosed in single quotes, such as `'A', '+', or '\n'`. In the mikroC for dsPIC30/33 and PIC24, single-character constants are of the unsigned int type. Multi-character constants are referred to as *string constants* or *string literals*. For more information refer to String Constants.

Escape Sequences

A backslash character (`\`) is used to introduce an escape sequence, which allows a visual representation of certain nongraphic characters. One of the most common escape constants is the newline character (`\n`).

A backslash is used with octal or hexadecimal numbers to represent an ASCII symbol or control code corresponding to that value; for example, '`\x3F`' for the question mark. Any value within legal range for data type char (`0 to 0xFF` for the mikroC for dsPIC30/33 and PIC24) can be used. Larger numbers will generate the compiler error "Out of range".

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

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

The following table shows the available escape sequences in mikroC:

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

String Constants

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

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

The *null string*, or empty string, is written like "". A literal string is stored internally as a given sequence of characters plus a final null character. A null string is stored as a single null character. The characters inside the double quotes can include escape sequences. This code, for example:

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

prints like this:

```
        "Name"\       Address
```

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

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

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

is an equivalent to

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

Line continuation with backslash

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

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

Enumeration Constants

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

For example:

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

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

Pointer Constants

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

Consider the following examples:

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

The following assignments are legal:

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

The following assignments are illegal:

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

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

Constant Expressions

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

Constant expressions can consist only of the following:

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

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

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

Constant expression can be used anywhere a constant is legal.

KEYWORDS

Keywords are words reserved for special purposes and must not be used as normal identifier names.

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

Here is an alphabetical listing of keywords in C:

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

Also, the mikroC for dsPIC30/33 and PIC24 includes a number of predefined identifiers used in libraries. You could replace them by your own definitions, if you want to develop your own libraries. For more information, see mikroC for dsPIC30/33 and PIC24 Libraries.

IDENTIFIERS

Identifiers are arbitrary names of any length given to functions, variables, symbolic constants, user-defined data types, and labels. All these program elements will be referred to as objects throughout the help (don't get confused with the meaning of object in object-oriented programming). Identifiers can contain the letters a to z and A to Z, underscore character "_", and digits 0 to 9. The only restriction is that the first character must be a letter or an underscore.

Case Sensitivity

The mikroC for dsPIC30/33 and PIC24 identifiers are not case sensitive at present, so that `Sum`, `sum`, and `suM` represent an equivalent identifier. However, future versions of the mikroC for dsPIC will offer an option of activating/suspending case sensitivity. The only exceptions at present are the reserved words `main` and `interrupt` which must be written in lower case.

Uniqueness and Scope

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

Identifier Examples

Here are some valid identifiers:

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

… and here are some invalid identifiers:

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

PUNCTUATORS

The mikroC for dsPIC30/33 and PIC24 punctuators (also known as separators) are:
[ ] – Brackets
( ) – Parentheses
{ } – Braces
, – Comma
; – Semicolon
: – Colon
* – Asterisk
= – Equal sign
– Pound sign

Most of these punctuators also function as operators.

Brackets

Brackets [ ] indicate single and multidimensional array subscripts:

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

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

Parentheses

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

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

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

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

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

Braces

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

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

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

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

For more information, refer to Compound Statements.

Comma

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

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

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

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

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

Semicolon

Semicolon (;) is a statement terminator. Any legal C expression (including the empty expression) followed by a semicolon is interpreted as a statement, known as an expression statement. The expression is evaluated and its value is discarded. If the expression statement has no side effects, the mikroC for dsPIC30/33 and PIC24 might ignore it.

```
a + b;        /* evaluate a + b, but discard value */
++a;          /* side effect on a, but discard value of ++a */
;             /* empty expression or a null statement */
```

Semicolons are sometimes used to create an empty statement:

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

For more information, see Statements.

Colon

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

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

Labels are discussed in Labeled Statements.

Asterisk (Pointer Declaration)

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

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

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

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

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

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

For more information, see Pointers.

Equal Sign

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

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

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

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

For more information, see Assignment Operators.

Pound Sign (Preprocessor Directive)

Pound sign (#) indicates a preprocessor directive when it occurs as the first non-whitespace character on a line. It signifies a compiler action, not necessarily associated with a code generation. See the Preprocessor Directives for more information. # and ## are also used as operators to perform token replacement and merging during the preprocessor scanning phase. See the Preprocessor Operators.

CONCEPTS

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

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

OBJECTS AND LVALUES

Objects

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

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

Objects and Declarations

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

Associating identifiers with objects requires each identifier to have at least two attributes: storage class and type (sometimes referred to as data type). The mikroC for dsPIC30/33 and PIC24 compiler deduces these attributes from implicit or explicit declarations in the source code. Usually, only the type is explicitly specified and the storage class specifier assumes the automatic value auto.

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

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

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

Lvalues

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

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

Rvalues

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

SCOPE AND VISIBILITY

Scope

The scope of an identifier is a part of the program in which the identifier can be used to access its object. There are different categories of scope: block (or local), function, function prototype, and file. These categories depend on how and where identifiers are declared.

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

File
- File scope identifiers, also known as globals, are declared outside of all blocks; their scope is from the point of declaration to the end of the source file.

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

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

Visibility

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

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

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

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

   {               // nested block
    double j;     // j is local name in the nested block
    j = 0.1;      // i and double j are visible;
                   // int j = 3 in scope but hidden
   }
                   // double j out of scope
 j += 1;           // int j visible and = 4
}
// i and j are both out of scope
```

NAME SPACES

Name space is a scope within which an identifier must be unique. The mikroC for dsPIC30/33 and PIC24 uses four distinct categories of identifiers:

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

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

For example:

```
int blue = 73;

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

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

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

DURATION

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

Static Duration

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

In the mikroC for dsPIC30/33 and PIC24, static duration objects are *not* initialized to zero (or null) in the absence of any explicit initializer.

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

Local Duration

Local duration objects are also known as *automatic* objects. They are created on the stack (or in a register) when an enclosing block or a function is entered. They are deallocated when the program exits that block or function. Local duration objects must be explicitly initialized; otherwise, their contents are unpredictable.

The storage class specifier auto can be used when declaring local duration variables, but it is usually redundant, because auto is default for variables declared within a block.

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

For example:

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

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

TYPES

The mikroC for dsPIC30/33 and PIC24 is a strictly typed language, which means that every object, function, and expression must have a strictly defined type, known in the time of compilation. Note that the mikroC for dsPIC30/33 and PIC24 works exclusively with numeric types. The type serves:

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

The mikroC for dsPIC30/33 and PIC24 supports many standard (predefined) and user-defined data types, including signed and unsigned integers in various sizes, floating-point numbers with various precisions, arrays, structures, and unions. In addition, pointers to most of these objects can be established and manipulated in memory.

The type determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object's storage allocation. A given data type can be viewed as a set of values (often implementation-dependent) that identifiers of that type can assume, together with a set of operations allowed with these values. The compile-time operator sizeof allows you to determine the size in bytes of any standard or user-defined type.
The mikroC for dsPIC30/33 and PIC24 standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that the mikroC for dsPIC can consistently access, interpret, and (possibly) change the bit patterns in memory corresponding to each active object in your program.

Type Categories

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

- fundamental
- derived

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

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

FUNDAMENTAL TYPES

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

The fundamental types include:

- Arithmetic Types
- Enumerations
- Void Type

Arithmetic Types

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

Integral Types

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

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

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

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

Below is the overview of arithmetic types:

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

Floating-point Types

The types float and double, together with the long double variant, are considered to be floating-point types. The mikroC for dsPIC30/33 and PIC24's implementation of an ANSI Standard considers all three to be the same type.

Floating point in the mikroC for dsPIC30/33 and PIC24 is implemented using the Microchip AN575 32-bit format (IEEE 754 compliant).

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

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

Enumerations

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

Enumeration Declaration

Enumeration is declared like this:

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

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

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

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

establishes a unique integral type, `enum` colors, variable c of this type, and set of enumerators with constant integer values (black = 0, red = 1, ...). In the mikroC for dsPIC30/33 and PIC24, a variable of an enumerated type can be assigned any value of the type `int` – no type checking beyond that is enforced. That is:

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

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

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

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

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

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

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

Anonymous Enum Type

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

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

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

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

Enumeration Scope

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

```
int blue = 73;

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

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

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

Void Type

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

Void Functions

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

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

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

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

Generic Pointers

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

DERIVED TYPES

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

- arrays
- pointers
- structures
- unions

Arrays

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

Array Declaration

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

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

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

Each of the elements of an array is numbered from 0 through the number of elements minus one. If the number is n, elements of array can be approached as variables *array_name*[0] .. *array_name*[n-1] of *type*.

Here are a few examples of array declaration:

```
#define MAX = 50

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

Array Initialization

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

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

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

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

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

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

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

Arrays in Expressions

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

Multi-dimensional Arrays

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

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

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

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

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

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

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

void main() {
  //...
  func(a);
}//~!
```

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

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

Pointers

Pointers are special objects for holding (or "pointing to") memory addresses. In the mikroC for dsPIC30/33 and PIC24, address of an object in memory can be obtained by means of an unary operator &. To reach the pointed object, we use an indirection operator (*) on a pointer.

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

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

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

Pointer Declarations

Pointers are declared the same as any other variable, but with * ahead of identifier. A type at the beginning of declaration specifies the type of a pointed object. A pointer must be declared as pointing to some particular type, even if that type is void, which really means a pointer to anything. Pointers to void are often called generic pointers, and are treated as pointers to char in the mikroC for dsPIC30/33 and PIC24.

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

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

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

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

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

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

/* is same as: */

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

Once declared, though, a pointer can usually be reassigned so that it points to an object of another type. The mikroC for dsPIC30/33 and PIC24 lets you reassign pointers without typecasting, but the compiler will warn you unless the pointer was originally declared to be pointing to `void`. You can assign the `void*` pointer to the `non-void*` pointer – refer to void for details.

Null Pointers

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

For example:

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

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

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

```
void *vp;
```

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

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

Function Pointers

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

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

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

Assign an address to a Function Pointer

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

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

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

Example:

```
int addC(char x,char y){

  return x+y;
}

int subC(char x,char y){

  return x-y;
}

int mulC(char x,char y){

  return x*y;
}


int divC(char x,char y){

  return x/y;
}


int modC(char x,char y){

  return x%y;
}


//array of pointer to functions that receive two chars and returns
int

int  (*arrpf[])(char,char) = {  addC ,subC,mulC,divC,modC};


int res;
char i;
void main() {

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

}//~!
```

Pointer Arithmetic

Pointer arithmetic in C is limited to:

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

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

Arrays and Pointers

Arrays and pointers are not completely independent types in the mikroC for dsPIC30/33 and PIC24. When the name of an array comes up in expression evaluation (except with operators `&` and `sizeof`), it is implicitly converted to the pointer pointing to array's first element. Due to this fact, arrays are not modifiable lvalues.

Brackets `[ ]` indicate array subscripts. The expression

```
id[ exp]
```

is defined as

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

where either:

id is a pointer and *exp* is an integer, or
id is an integer and *exp* is a pointer.

The following is true:

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

According to these guidelines, we can write:

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

Also, you need to be careful with operator precedence:

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

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

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

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

Assignment and Comparison

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

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

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

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

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

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

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

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

Pointer Addition

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

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

For example:

```
int a[10];          // array a containing 10 elements of int
int *pa = &a[0];    // pa is pointer to int, pointing to a[0]

*(pa + 3) = 6;      // pa+3 is a pointer pointing to a[3],
                    // so a[3] now equals 6
pa++; // pa now points to the next element of array, a[1]
```

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

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

For example:

```
void f (some_type a[], int n) {
     /* function f handles elements of array a; */
     /* array a has n elements of some_type */

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

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

Pointer Subtraction

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

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

For example:

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

Structures

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

Unlike arrays, structures are considered to be single objects. The mikroC for dsPIC30/33 and PIC24 structure type lets you handle complex data structures almost as easily as single variables.

Note: the mikroC for dsPIC30/33 and PIC24 does not support anonymous structures (ANSI divergence).

Structure Declaration and Initialization

Structures are declared using the keyword `struct`:

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

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

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

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

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

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

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

Note that you can omit structure tag, but then you cannot declare additional objects of this type elsewhere. For more information, see the "Untagged Structures" below.

Structure is initialized by assigning it a comma-delimited sequence of values within braces, similar to array. Referring to declarations from the previous example:

```
/* Referring to declarations from the example above: */
/* Declare and initialize dots p and q: */
struct Dot p = {1., 1.}, q = {3.7, -0.5};

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

Incomplete Declarations

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

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

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

Untagged Structures and Typedefs

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

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

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

Usually, there is no need to use both `tag` and `typedef`: either can be used in structure type declarations. Untagged structure and union members are ignored during initialization.
Note: See also Working with structures.

Working with Structures

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

Assignment

Variables of the same structured type may be assigned one to another by means of simple assignment operator (=). This will copy the entire contents of the variable to destination, regardless of the inner complexity of a given structure.

Note that two variables are of the same structured type only if they are both defined by the same instruction or using the same type identifier. For example:

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

Size of Structure

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

Structures and Functions

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

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

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

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

Structure Member Access

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

. (period)
-> (right arrow)

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

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

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

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

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

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

For example:

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

...

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

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

Accessing Nested Structures

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

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

//...

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

Structure Uniqueness

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

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

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

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

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

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

Unions

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

Note: The mikroC for dsPIC30/33 and PIC24 does not support anonymous unions (ANSI divergence).

Union Declaration

Unions are declared same as structures, with the keyword `union` used instead of `struct`:

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

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

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

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

Size of Union

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

Union Member Access

Union members can be accessed with the structure member selectors (`.` and `->`), be careful when doing this. Check the example on the following page.

Referring to declarations from the previous example:

```
pm = &mu;
mu.d = 4.016;
tmp = mu.d;  // OK: mu.d = 4.016
tmp = mu.i;  // peculiar result

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

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

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

Bit Fields

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

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

You cannot take the address of a bit field.

Note: If you need to handle specific bits of 8-bit variables (char and unsigned short) or registers, you don't need to declare bit fields. Much more elegant solution is to use the mikroC for dsPIC30/33 and PIC24's intrinsic ability for individual bit access — see Accessing Individual Bits for more information.

Bit Fields Declaration

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

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

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

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

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

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

Here is an example:

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

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

Bit Fields Access

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

```
// Declare a bit field Value_For_PortB:
myunsigned Value_For_PortB;

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

TYPES CONVERSIONS

The mikroC for dsPIC30/33 and PIC24 is a strictly typed language, with each operator, statement and function demanding appropriately typed operands/arguments. However, we often have to use objects of “mismatching” types in expressions. In that case, *type conversion* is needed.

Conversion of object of one type means that object's type is changed into another type. The mikroC for dsPIC30/33 and PIC24 defines a set of standard conversions for built-in types, provided by compiler when necessary. For more information, refer to the Standard Conversions.

Conversion is required in following situations:

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

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

Standard Conversions

Standard conversions are built in the mikroC for dsPIC30/33 and PIC24. These conversions are performed automatically, whenever required in the program. They can also be explicitly required by means of the typecast operator (refer to the Explicit Typecasting).

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

Arithmetic Conversions

When using arithmetic expression, such as a + b, where a and b are of different arithmetic types, the mikroC for dsPIC30/33 and PIC24 performs implicit type conversions before the expression is evaluated. These standard conversions include promotions of “lower” types to “higher” types in the interests of accuracy and consistency.

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

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

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

In details:

Here are the steps the mikroC for dsPIC30/33 and PIC24 uses to convert the operands in an arithmetic expression:

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

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

After this, any two values associated with an operator are either int (including the long and unsigned modifiers) or float (equivalent with double and long double in the mikroC for dsPIC30/33 and PIC24).

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

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

Here are several examples of implicit conversion:

```
2+3.1          // = 2. + 3.1 = 5.1
5/4*3.         // = (5/4)*3. = 1*3. = 1.*3. = 3.0
3.*5/4         // = (3.*5)/4 = (3.*5.)/4 = 15./4 = 15./4. = 3.75
```

Pointer Conversions

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

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

More generally, the cast (*type**) will convert a pointer to type "pointer to *type*".

Explicit Types Conversions (Typecasting)

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

```
(type) object
```

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

For example:

```
char a, b;

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

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

DECLARATIONS

Introduction to Declarations

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

The range of objects that can be declared includes:

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

Declarations and Definitions

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

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

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

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

Let's have an example:

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

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

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

Declarations and Declarators

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

Declarations of variable identifiers have the following pattern:

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

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

For example:

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

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

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

Linkage

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

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

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

Linkage Rules

Local names have internal linkage; the same identifier can be used in different files to signify different objects. Global names have external linkage; identifier signifies the same object throughout all program files.

If the same identifier appears with both internal and external linkage within the same file, the identifier will have internal linkage.

Internal Linkage Rules:

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

External Linkage Rule:

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

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

Storage Classes

Associating identifiers with objects requires each identifier to have at least two attributes: storage class and type (sometimes referred to as data type). The mikroC for dsPIC30/33 and PIC24 compiler deduces these attributes from implicit or explicit declarations in the source code.

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

```
storage-class type identifier
```

The storage class specifiers in the mikroC for dsPIC30/33 and PIC24 are:

```
auto
register
static
extern
```

Auto

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

Register

At the moment the modifier `register` technically has no special meaning. The mikroC for dsPIC30/33 and PIC24 compiler simply ignores requests for register allocation..

Static

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

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

Extern

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

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

See *Linkage* for more information.

Type Qualifiers

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

Qualifier const

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

The mikroC for dsPIC30/33 and PIC24 treats objects declared with the const qualifier the same as literals or preprocessor constants. If the user tries to change an object declared with the const qualifier compiler will report an error.

For example:

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

Qualifier volatile

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

Typedef Specifier

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

The specifier typedef stands first in the declaration:

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

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

A declaration starting with the typedef specifier does not introduce an object or a function of a given type, but rather a new name for a given type. In other words, the typedef declaration is identical to a “normal” declaration, but instead of objects, it declares types. It is a common practice to name custom type identifiers with starting capital letter — this is not required by the mikroC for dsPIC30/33 and PIC24.

For example:

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

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

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

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

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

asm Declaration

The mikroC for dsPIC30/33 and PIC24 allows embedding assembly in the source code by means of the `asm` declaration. The declarations `_asm` and `__asm` are also allowed in the mikroC for dsPIC30/33 and PIC24 and have the same meaning. Note that numerals cannnot be used as absolute addresses for SFR or GPR variables in assembly instructions. Symbolic names may be used instead (listing will display these names as well as addresses).

Assembly instructions can be grouped by the asm keyword (or `_asm`, or `__asm`):

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

The mikroC for dsPIC30/33 and PIC24 comments (both single-line and multi-line) are allowed in embedded assembly code.

If you plan to use a certain C variable in embedded assembly only, make sure that it will not be eliminated by optimization process; otherwise, linker will issue an error. This does not apply to predefined globals such as PORTB.

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

```
unsigned myvar;
void main() {
  asm {
    MOVLW 10   // just a test
    MOVLW _myvar
    MOVLW 0    // just a test
    MOVLW _myvar+1
  }
}
```

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

```
if (myvar)
  ;
```

The syntax that is being used in the asm blocks is a bit different from that in version 2.0.0.0.

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

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

The only types whose name remains the same in asm as it is in the mikroC for dsPIC30/33 and PIC24 are constants, e.g. INTCON, PORTB, WREG, GIE, etc.

Accessing individual bytes is different as well. For example, a global variable "g_var" of type long (i.e. 4 bytes) can be accessed like this:

```
MOVF    _g_var+0, 0
;puts least-significant byte of g_var in W register

MOVF    _g_var+1, 0
;second byte of _g_var; corresponds to Hi(g_var)

MOVF    _g_var+2, 0     ;Higher(g_var)
MOVF    _g_var+3, 0     ;Highest(g_var)
;... etc.
```

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

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

For objects located in RAM:

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

Initialization

The initial value of a declared object can be set at the time of declaration (*initialization*). A part of the declaration which specifies the initialization is called *initializer*.

Initializers for globals and `static` objects must be constants or constant expressions. The initializer for an automatic object can be any legal expression that evaluates to an assignment-compatible value for the type of the variable involved.

Scalar types are initialized with a single expression, which can optionally be enclosed in braces. The initial value of an object is that of the expression; the same constraints for type and conversions as for simple assignments are applied to initializations too.

For example:

```
int i = 1;
char *s = "hello";
struct complex c = {0.1, -0.2};
// where 'complex' is a structure (float, float)
```

For structures or unions with automatic storage duration, the initializer must be one of the following:

- an initializer list,
- a single expression with compatible union or structure type. In this case, the initial value of the object is that of the expression.

For more information, refer to Structures and Unions.

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

Automatic Initialization

The mikroC for dsPIC30/33 and PIC24 does not provide automatic initialization for objects. Uninitialized globals and objects with static duration will take random values from memory.

FUNCTIONS

Functions are central to C programming. Functions are usually defined as subprograms which return a value based on a number of input parameters. Return value of the function can be used in expressions – technically, function call is considered to be an expression like any other.

C allows a function to create results other than its return value, referred to *as side effects*. Often, the function return value is not used at all, depending on the side effects. These functions are equivalent to *procedures* of other programming languages, such as Pascal. C does not distinguish between procedure and function – functions play both roles.

Each program must have a single external function named main marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions have external linkage by default and are normally accessible from any file in the program. This can be restricted by using the `static` storage class specifier in function declaration (see Storage Classes and Linkage).

Note: Check the dsPIC30/33 and PIC24 Specifics for more information on functions' limitations on the dsPIC30/33 and PIC24 micros.

Function Declaration

Functions are declared in user's source files or made available by linking precompiled libraries. The declaration syntax of the function is:

```
type function_name(parameter-declarator-list);
```

The `function_name` must be a valid identifier. This name is used to call the function; see Function Calls for more information.

`type` represents the type of function result, and can be of any standard or user-defined type. For functions that do not return value the void type should be used. The type can be omitted in global function declarations, and function will assume the int type by default.

Function type can also be a pointer. For example, *float** means that a function result is a pointer to float. The generic pointer *void** is also allowed.

The function *cannot* return an array or another function.

Within parentheses, *parameter-declarator-list* is a list of formal arguments that function takes. These declarators specify the type of each function parameter. The compiler uses this information to check validity of function calls. If the list is empty, a function does not take any arguments. Also, if the list is `void`, a function also does not take any arguments; note that this is the only case when void can be used as an argument's type.

Unlike variable declaration, each argument in the list needs its own type specifier and possible qualifier `const` or `volatile`.

Function Prototypes

A function can be defined only once in the program, but can be declared several times, assuming that the declarations are compatible. When declaring a function, the formal argument's identifier does not have to be specified, but its type does.

This kind of declaration, commonly known as the *function prototype*, allows better control over argument number, type checking and type conversions. The name of a parameter in function prototype has its scope limited to the prototype. This allows one parameter identifier to have different name in different declarations of the same function:

```
/* Here are two prototypes of the same function: */

int test(const char*)    // declares function test
int test(const char*p)   // declares the same function test
```

Function prototypes are very useful in documenting code. For example, the function `Cf_Init` takes two parameters: Control Port and Data Port. The question is, which is which? The function prototype:

```
void Cf_Init(char *ctrlport, char *dataport);
```

makes it clear. If a header file contains function prototypes, the user can read that file to get the information needed for writing programs that call these functions. If a prototype parameter includes an identifier, then the indentifier is only used for error checking.

Function Definition

Function definition consists of its declaration and *function body*. The function body is technically a block – a sequence of local definitions and statements enclosed within braces {}. All variables declared within function body are local to the function, i.e. they have function scope.

The function itself can be defined only within the file scope, which means that function declarations cannot be nested.

To return the function result, use the return statement. The statement return in functions of the void type cannot have a parameter – in fact, the return statement can be omitted altogether if it is the last statement in the function body.

Here is a sample function definition:

```
/* function max returns greater one of its 2 arguments: */

int max(int x, int y) {
  return (x>=y) ? x : y;
}
```

Here is a sample function which depends on side effects rather than return value:

```
/* function converts Descartes coordinates (x,y)
   to polar coordinates (r,fi): */

#include <math.h>

void polar(double x, double y, double *r, double *fi) {
  *r = sqrt(x * x + y * y);
  *fi = (x == 0 && y == 0) ? 0 : atan2(y, x);
  return; /* this line can be omitted */
}
```

Function Reentrancy

Functions reentrancy is allowed. Remember that the dsPIC's and PIC24 has stack and memory limitations which can varies greatly between MCUs.

Function Calls

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

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

Each `expression` in the function call is an *actual argument*. Number and types of actual arguments should match those of formal function parameters. If types do not match, implicit type conversions rules will be applied. Actual arguments can be of any complexity, but order of their evaluation is not specified.

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

If the function has no result (type `void`) or the result is not needed, then the function call can be written as a self-contained expression.

In C, scalar arguments are always passed to the function by value. The function can modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine. A scalar object can be passed by the address if a formal parameter is declared as a pointer. The pointed object can be accessed by using the indirection operator *.

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

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

Argument Conversions

If a function prototype has not been previously declared, the mikroC for dsPIC30/33 and PIC24 converts integral arguments to a function call according to the integral widening (expansion) rules described in Standard Conversions. If a function prototype is in scope, the mikroC for dsPIC30/33 and PIC24 converts the passed argument to the type of the declared parameter according to the same conversion rules as in assignment statements.

If a prototype is present, the number of arguments must match. The types need to be compatible only to the extent that an assignment can legally convert them. The user can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

Note: If the function prototype does not match the actual function definition, the mikroC for dsPIC30/33 and PIC24 will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with the corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and actual definitions will be detected.

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

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

extern long func(long par1, long par2);

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

Since the program has the function prototype for `func`, it converts `limit` and `ch` to `long`, using the standard rules of assignment, before it places them on the stack for the call to `func`.

Without the function prototype, `limit` and `ch` would be placed on the stack as an integer and a character, respectively; in that case, the stack passed to `func` will not match size or content that `func` expects, which can cause problems.

Ellipsis ('...') Operator

The ellipsis ('...') consists of three successive periods with no whitespace intervening. An ellipsis can be used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types. For example:

```
void func (int n, char ch, ...);
```

This declaration indicates that func will be defined in such a way that calls must have at least two arguments, int and char, but can also have any number of additional arguments.

Example:

```
#include <stdarg.h>

int addvararg(char a1,...){
va_list ap;
char temp;
va_start(ap,a1);

while( temp = va_arg(ap,char))
  a1 += temp;
return a1;
}

int res;
void main() {

  res = addvararg(1,2,3,4,5,0);

  res = addvararg(1,2,3,4,5,6,7,8,9,10,0);

} //~!
```

OPERATORS

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

- Arithmetic Operators
- Assignment Operators
- Bitwise Operators
- Logical Operators
- Reference/Indirect Operators
- Relational Operators
- Structure Member Selectors
- Comma Operator `,`
- Conditional Operator `? :`
- Array subscript operator `[]`
- Function call operator `()`
- `sizeof` Operator
- Preprocessor Operators `#` and `##`

Operators Precedence and Associativity

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

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

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

Arithmetic Operators

Arithmetic operators are used to perform mathematical computations. They have numerical operands and return numerical results. The type technically represents small integers, so the char variables can be used as operands in arithmetic operations. All arithmetic operators associate from left to right.

Operator	Operation	Precedence
Binary Operators		
+	addition	12
-	subtraction	12
*	multiplication	13
/	division	13
%	modulus operator returns the remainder of integer division (cannot be used with floating points)	13
Unary Operators		
+ (unary)	unary plus does not affect the operand	14
- (unary)	unary minus changes the sign of operand	14
++	increment adds one to the value of the operand. Postincrement adds one to the value of the operand after it evaluates; while preincrement adds one before it evaluates	14
--	decrement subtracts one from the value of the operand. Postdecrement subtracts one from the value of the operand after it evaluates; while predecrement subtracts one before it evaluates.	14

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

Binary Arithmetic Operators

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

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

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

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

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

We can use arithmetic operators for manipulating characters:

```
'A' + 32;           // equals 'a' (ASCII only)
'G' - 'A' + 'a';    // equals 'g' (both ASCII and EBCDIC)
```

Unary Arithmetic Operators

Unary operators ++ and -- are the only operators in C which can be either prefix (e.g. ++k, --k) or postfix (e.g. k++, k--).

When used as prefix, operators ++ and -- (preincrement and predecrement) add or subtract one from the value of operand *before* the evaluation. When used as suffix, operators ++ and -- add or subtract one from the value of operand *after* the evaluation.

For example:

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

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

Relational Operators

Use relational operators to test equality or inequality of expressions. If the expression evaluates to true, it returns 1; otherwise it returns 0.
All relational operators associate from left to right.

Relational Operators Overview

Operator	Operation	Precedence
==	equal	9
!=	not equal	9
>	greater than	10
<	less than	10
>=	greater than or equal	10
<=	less than or equal	10

Relational Operators in Expressions

Precedence of arithmetic and relational operators was designated in such a way to allow complex expressions without parentheses to have expected meaning:

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

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

```
/* ok: */
5 > 7                    /* returns 0 */
10 <= 20                 /* returns 1 */
/* this can be tricky: */
8 == 13 > 5              /* returns 0, as: 8 == (13 > 5)  -> 8 == 1
                            ->  0 */
14 > 5 < 3               /* returns 1, as: (14 > 5) < 3  ->  1 < 3
                            ->  1 */
a < b < 5                /* returns 1, as: (a < b) < 5  -> (0 or 1)
                             < 5  ->  1*/
```

Bitwise Operators

Use the bitwise operators to modify the individual bits of numerical operands.

Bitwise operators associate from left to right. The only exception is the bitwise complement operator ~ which associates from right to left.

Bitwise Operators Overview

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

Logical Operations on Bit Level

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

~	0	1
	1	0

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

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

/* because ..

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

.. that is, 0x1230 */


/* Similarly: */

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

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

Bitwise Shift Operators

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

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

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

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

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

Bitwise vs. Logical

Be aware of the principle difference between how bitwise and logical operators work. For example:

```
0222222 &  0555555;       /* equals 000000 */
0222222 && 0555555;       /* equals 1 */

~ 0x1234;                 /* equals 0xEDCB */
! 0x1234;                 /* equals 0 */
```

Logical Operators

Operands of logical operations are considered true or false, that is non-zero or zero. Logical operators always return 1 or 0. Operands in a logical expression must be of scalar type.

Logical operators && and || associate from left to right. Logical negation operator ! associates from right to left.

Logical Operators Overview

Operator	Operation	Precedence
&&	logical AND	5
||	logical OR	4
!	logical negation	14

&&	0	1
0	0	0
1	0	1

||	0	1
0	0	1
1	1	1

!	0	1
	1	0

Precedence of logical, relational, and arithmetic operators was designated in such a way to allow complex expressions without parentheses to have an expected meaning:

```
c >= '0' && c <= '9'; // reads as: (c >= '0') && (c <= '9')
a + 1 == b || ! f(x); // reads as: ((a + 1) == b) || (! (f(x)))
```

Logical AND && returns 1 only if both expressions evaluate to be nonzero, otherwise returns 0. If the first expression evaluates to false, the second expression will not be evaluated. For example:

```
a > b && c < d;      // reads as  (a > b) && (c < d)
// if (a > b) is false (0), (c < d) will not be evaluated
```

Logical OR `||` returns 1 if either of expression evaluates to be nonzero, otherwise returns 0. If the first expression evaluates to true, the second expression is not evaluated. For example:

```
a && b || c && d;  /* reads as: (a && b) || (c && d) */
/* if (a && b) is true (1), (c && d) will not be evaluated */
```

Logical Expressions and Side Effects

General rule regarding complex logical expressions is that the evaluation of consecutive logical operands stops at the very moment the final result is known. For example, if we have an expression a `&&` b `&&` c where a is false (0), then operands b and c will not be evaluated. This is very important if `b` and `c` are expressions, as their possible side effects will not take place!

Logical vs. Bitwise

Be aware of the principle difference between how bitwise and logical operators work. For example

```
0222222 &  0555555        /* equals 000000 */
0222222 && 0555555        /* equals 1 */

~ 0x1234                  /* equals 0xEDCB */
! 0x1234                  /* equals 0 */
```

Conditional Operator ? :

The conditional operator ? : is the only ternary operator in C. Syntax of the conditional operator is:

```
expression1 ? expression2 : expression3
```

Expression1 evaluates first. If its value is true, then *expression2* evaluates and *expression3* is ignored. If *expression1* evaluates to false, then *expression3* evaluates and *expression2* is ignored. The result will be the value of either *expression2* or *expression3* depending upon which evaluates. The fact that only one of these two expressions evaluates is very important if you expect them to produce side effects!

Conditional operator associates from right to left.

Here are a couple of practical examples:

```
/* Find max(a, b): */
max = (a > b) ? a : b;

/* Convert small letter to capital: */
/* (no parentheses are actually necessary) */
c = (c >= 'a' && c <= 'z') ? (c - 32) : c;
```

Conditional Operator Rules

Expression1 must be a scalar expression; *expression2* and *expression3* must obey one of the following rules:

1. Both expressions have to be of arithmetic type. *expression2* and *expression3* are subject to usual arithmetic conversions, which determines the resulting type.

2. Both expressions have to be of compatible `struct` or `union` types. The resulting type is a structure or union type of *expression2* and *expression3*.

3. Both expressions have to be of `void` type. The resulting type is `void`.

4. Both expressions have to be of type pointer to qualified or unqualified versions of compatible types. The resulting type is a pointer to a type qualified with all type qualifiers of the types pointed to by both expressions.
5. One expression is a pointer, and the other is a null pointer constant. The resulting type is a pointer to a type qualified with all type qualifiers of the types pointed to by both expressions.
6. One expression is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of `void`. The resulting type is that of the `non-pointer-to-void` expression.

Assignment Operators

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

Simple Assignment Operator

For a common value assignment, a simple assignment operator (=) is used:

```
expression1 = expression2
```

The *expression1* is an object (memory location) to which the value of *expression2* is assigned. Operand *expression1* has to be lvalue and *expression2* can be any expression. The assignment expression itself is not lvalue.

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

Compound Assignment Operators

C allows more comlex assignments by means of compound assignment operators. The syntax of compound assignment operators is:

```
expression1 op= expression2
```

where *op* can be one of binary operators `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, or `>>`.

Thus, we have 10 different compound assignment operators: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, and `>>=`. All of these associate from right to left. Spaces separating compound operators (e.g. `+ =`) will generate error.

Compound assignment has the same effect as

```
expression1 = expression1 op expression2
```

except the lvalue *expression1* is evaluated only once. For example,

```
expression1 += expression2
```

is the same as

```
expression1 = expression1 + expression2
```

Assignment Rules

For both simple and compound assignment, the operands expression1 and expression2 must obey one of the following rules:

1. *expression1* is of qualified or unqualified arithmetic type and *expression2* is of arithmetic type.
2. *expression1* has a qualified or unqualified version of structure or union type compatible with the type of *expression2*.
3. *expression1* and *expression2* are pointers to qualified or unqualified versions of compatible types and the type pointed to by left has all qualifiers of the type pointed to by right.
4. Either *expression1* or *expression2* is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of void. The type pointed to by left has all qualifiers of the type pointed to by right.
5. *expression1* is a pointer and *expression2* is a null pointer constant.

Sizeof Operator

The prefix unary operator `sizeof` returns an integer constant that represents the size of memory space (in bytes) used by its operand (determined by its type, with some exceptions). The operator `sizeof` can take either a type identifier or an unary expression as an operand. You *cannot* use `sizeof` with expressions of function type, incomplete types, parenthesized names of such types, or with lvalue that designates a bit field object.

Sizeof Applied to Expression

If applied to expression, the `sizeof` an operand is determined without evaluating the expression (and therefore without side effects). The result of the operation will be the size of the type of the expression's result.

Sizeof Applied to Type

If applied to a type identifier, sizeof returns the size of the specified type. The unit for type size is `sizeof(char)` which is equivalent to one byte.
The operation `sizeof(char)` gives the result 1, whether `char` is `signed` or `unsigned`.

```
sizeof(char)                /* returns 1 */
sizeof(int)                 /* returns 2 */
sizeof(unsigned long)       /* returns 4 */
sizeof(float)               /* returns 4 */
```

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

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

If the operand is a parameter declared as array type or function type, `sizeof` gives the size of the pointer. When applied to structures and unions, `sizeof` gives the total number of bytes, including any padding. The operator `sizeof` cannot be applied to a function.

EXPRESSIONS

Expression is a sequence of operators, operands, and punctuators that specifies a computation. Formally, expressions are defined recursively: subexpressions can be nested without formal limit. However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.

In ANSI C, the *primary* expressions are: constant (also referred to as literal), identifier, and (`expression`), defined recursively.

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

Expressions can produce lvalue, rvalue, or no value. Expressions might cause side effects whether they produce a value or not.

Comma Expressions

One of the specifics of C is that it allows using of comma as a *sequence operator* to form so-called *comma expressions* or . Comma expression is a comma-delimited list of expressions – it is formally treated as a single expression so it can be used in places where an expression is expected. The following sequence:

```
expression_1, expression_2;
```

results in the left-to-right evaluation of each `expression`, with the value and type of `expression_2` giving the result of the whole expression. Result of `expression_1` is discarded.

Binary operator comma (,) has the lowest precedence and associates from left to right, so that `a, b, c` is same as `(a, b), c`. This allows us to write sequences with any number of expressions:

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

which results in the left-to-right evaluation of each `expression`, with the value and type of `expression_n` giving the result of the whole expression. Results of other `expressions` are discarded, but their (possible) side-effect do occur.

For example:

```
result = (a = 5, b /= 2, c++);
/* returns preincremented value of variable c, but also
   intializes a, divides b by 2, and increments c */

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

Note

Do not confuse comma operator (sequence operator) with comma punctuator which separates elements in a function argument list and initializator lists. To avoid ambiguity with commas in function argument and initializer lists, use parentheses. For example,

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

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

STATEMENTS

Statements specify a flow of control as the program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code.

Statements can be roughly divided into:

- Labeled Statements
- Expression Statements
- Selection Statements
- Iteration Statements (Loops)
- Jump Statements
- Compound Statements (Blocks)

Labeled Statements

Every statement in program can be labeled. Label is an identifier added before the statement like this:

```
label_identifier : statement;
```

There is no special declaration of a label – it just "tags" the statement. `Label_identifier` has a function scope and label cannot be redefined within the same function.

Labels have their own namespace: label identifier can match any other identifier in the program.

A statement can be labeled for two reasons:

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

2. The label identifier serves as a target for the `switch` statement. For this purpose, only `case` and `default` labeled statements are used:

```
case constant-expression : statement
default : statement
```

Expression Statements

Any expression followed by a semicolon forms an expression statement:

```
expression;
```

The mikroC for dsPIC30/33 and PIC24 executes an expression statement by evaluating the *expression*. All side effects from this evaluation are completed before the next statement starts executing. Most of expression statements are assignment statements or function calls.

A *null statement* is a special case, consisting of a single semicolon (;). The null statement does nothing, and therefore is useful in situations where the mikroC for dsPIC30/33 and PIC24 syntax expects a statement but the program does not need one. For example, a null statement is commonly used in "empty" loops:

```
for (; *q++ = *p++ ;);
/* body of this loop is a null statement */
```

Selection Statements

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements in C: `if` and `switch`.

If Statement

The `if` statement is used to implement a conditional statement. The syntax of the `if` statement is:

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

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

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

Nested if statements

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

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

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

Switch Statement

The `switch` statement is used to pass control to a specific program branch, based on a certain condition. The syntax of the `switch` statement is:

```
switch (expression) {
  case constant-expression_1 : statement_1;
     .
     .
     .
  case constant-expression_n : statement_n;
  [default : statement;]
}
```

First, the `expression` (condition) is evaluated. The switch statement then compares it to all available `constant-expressions` following the keyword `case`. If a match is found, switch passes control to that matching `case` causing the `statement` following the match evaluates. Note that `constant-expressions` must evaluate to integer. It is not possible to have two same constant expressions evaluating to the same value.

Parentheses around `expression` are mandatory.

.

Upon finding a match, program flow continues normally: the following instructions will be executed in natural order regardless of the possible `case` label. If no `case` satisfies the condition, the default `case` evaluates (if the label `default` is specified).

For example, if a variable `i` has value between 1 and 3, the following switch would always return it as 4:

```
switch (i) {
  case 1: i++;
  case 2: i++;
  case 3: i++;
}
```

To avoid evaluating any other cases and relinquish control from `switch`, each `case` should be terminated with break.

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

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

Nested switch

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

Iteration Statements

Iteration statements allows to loop a set of statements. There are three forms of iteration statements in the mikroC for dsPIC30/33 and PIC24:

- while
- do
- for

While Statement

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

```
while (expression) statement
```

The `statement` executes repeatedly until the value of `expression` is false. The test takes place before `statement` is executed. Thus, if `expression` evaluates to false on the first pass, the loop does not execute. Note that parentheses around `expression` are mandatory.

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

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

Note that body of a loop can be a null statement. For example:

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

Do Statement

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

```
do statement while (expression);
```

The *statement* is executed repeatedly as long as the value of *expression* remains non-zero. The *expression* is evaluated after each iteration, so the loop will execute *statement* at least once.

Parentheses around *expression* are mandatory.

Note that do is the only control structure in C which explicitly ends with semicolon (;). Other control structures end with statement which means that they implicitly include a semicolon or a closing brace.

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

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

For Statement

The for statement implements an iterative loop. Syntax of for statement is:

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

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

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

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

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

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

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

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

You can also do it like this:

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

but this is considered a bad programming style. Although legal, calculating the sum *should* not be a part of the incrementing expression, because it is not in the service of loop routine. Note that we used a null statement (`;`) for a loop body.

Jump Statements

A jump statement, when executed, transfers control unconditionally. There are four such statements in mikroC for dsPIC30/33 and PIC24: `break`, `continue`, `goto`, and `return`.

Break Statement

Sometimes it is necessary to stop the loop within its body. Use the `break` statement within loops to pass control to the first statement following the innermost `switch`, `for`, `while`, or `do` block.

`Break` is commonly used in `switch` statements to stop its execution upon the first positive match. For example:

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

Continue Statement

The `continue` statement within loops is used to "skip the cycle". It passes control to the end of the innermost enclosing end brace belonging to a looping construct. At that point the loop continuation condition is re-evaluated. This means that continue demands the next iteration if the loop continuation condition is true.

Specifically, the `continue` statement within the loop will jump to the marked position as it is shown below:

```
while (..) {
...
if (val>0) continue;
...
// continue jumps
here}
```

```
do {
...
if (val>0) continue;
...
// continue jumps
here
while (..);
```

```
for (..;..;..) {
...
if (val>0) continue;
...
// continue jumps
here
}
```

Goto Statement

The goto statement is used for unconditional jump to a local label — for more information on labels, refer to Labeled Statements. The syntax of the goto statement is:

```
goto label_identifier;
```

This will transfer control to the location of a local label specified by label_identifier. The label_identifier has to be a name of the label within the same function in which the goto statement is. The goto line can come before or after the label.

goto is used to break out from any level of nested control structures but it cannot be used to jump *into* block while skipping that block's initializations – for example, jumping into loop's body, etc.

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

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

Return Statement

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

```
return [ expression] ;
```

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

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

Compound Statements (Blocks)

The compound statement, or *block*, is a list (possibly empty) of statements enclosed in matching braces { }. Syntactically, the block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within the block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth up to the limits of memory.

For example, the for loop expects one statement in its body, so we can pass it a compound statement:

```
for (i = 0; i < n; i++) {
  int temp = a[ i] ;
  a[ i]  = b[ i] ;
  b[ i]  = temp;
}
```

Note that, unlike other statements, compound statements do not end with semicolon (`;`), i.e. there is never a semicolon following the closing brace.

PREPROCESSOR

Preprocessor is an integrated text processor which prepares the source code for compiling. Preprocessor allows:

- inserting text from a specifed file to a certain point in the code (see File Inclusion),
- replacing specific lexical symbols with other symbols (see Macros),
- conditional compiling which conditionally includes or omits parts of the code (see Conditional Compilation).

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

Preprocessor Directives

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

A *null directive* consists of a line containing the single character `#`. This line is always ignored.

Preprocessor directives are usually placed at the beginning of the source code, but they can legally appear at any point in a program. The mikroC for dsPIC30/33 and PIC24 preprocessor detects preprocessor directives and parses the tokens embedded in them. A directive is in effect from its declaration to the end of the program file.

Here is one commonly used directive:

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

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

The mikroC for dsPIC30/33 and PIC24 supports standard preprocessor directives:

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

Note: For the time being only funcall `#pragma` is supported.

Line Continuation with Backslash (\)

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

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

Macros

Macros provide a mechanism for a token replacement, prior to compilation, with or without a set of formal, function-like parameters.

Defining Macros and Macro Expansions

The `#define` directive defines a macro:

```
#define macro_identifier <token_sequence>
```

Each occurrence of `macro_identifier` in the source code following this control line will be replaced in the original position with the possibly empty `token_sequence` (there are some exceptions, which are discussed later). Such replacements are known as macro expansions.token_sequence is sometimes called the body of a macro. An empty token sequence results in the removal of each affected macro identifier from the source code.

No semicolon (`;`) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in a macro expansion.token_sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single-space character.

After each individual macro expansion, a further scan is made of the newly expanded text. This allows the possibility of using nested macros: the expanded text can contain macro identifiers that are subject to replacement. However, if the macro expands into something that looks like a preprocessing directive, such directive will not be recognized by the preprocessor. Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code will not be expanded.

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

Let's have an example:

```
/* Here are some simple macros: */
#define ERR_MSG "Out of range!"
#define EVERLOOP for( ; ; )

/* which we could use like this: */

main() {
  EVERLOOP {
    ...
    if (error) {Lcd_Out_Cp(ERR_MSG); break;}
    ...
  }
}
```

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is exactly the same token-by-token definition as the existing one. The preferred strategy where definitions might exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
  #define BLOCK_SIZE 512
#endif
```

The middle line is bypassed if `BLOCK_SIZE` is currently defined; if `BLOCK_SIZE` is not currently defined, the middle line is invoked to define it.

Macros with Parameters

The following syntax is used to define a macro with parameters:

```
#define macro_identifier(<arg_list>) token_sequence
```

Note that there can be no whitespace between `macro_identifier` and "`(`". The optional `arg_list` is a sequence of identifiers separated by commas, like the argument list of a C function. Each comma-delimited identifier has the role of a formal argument or placeholder.

Such macros are called by writing

```
macro_identifier(<actual_arg_list>)
```

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C "functions" are implemented as macros. However, there are some important semantic differences.

The optional `actual_arg_list` must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal `arg_list` of the `#define` line – there must be an actual argument for each formal argument. An error will be reported if the number of arguments in two lists is not the same.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in `actual_arg_list`. Like with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

Here is a simple example:

```
// A simple macro which returns greater of its 2 arguments:
#define _MAX(A, B) ((A) > (B)) ? (A) : (B)

// Let's call it:
x = _MAX(a + b, c + d);

/* Preprocessor will transform the previous line into:
x = ((a + b) > (c + d)) ? (a + b) : (c + d) */
```

It is highly recommended to put parentheses around each argument in the macro body in order to avoid possible problems with operator precedence.

Undefining Macros

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

```
#undef macro_identifier
```

Directive `#undef` detaches any previous token sequence from the `macro_identifier`; the macro definition has been forgotten, and the `macro_identifier` is undefined. No macro expansion occurs within `#undef` lines.

The state of being defined or undefined is an important property of an identifier, regardless of the actual definition. The `#ifdef` and `#ifndef` conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with `#define`, using the same or a different token sequence.

File Inclusion

The preprocessor directive `#include` pulls in header files (extension .h) into the source code. Do not rely on preprocessor to include source files (extension .c) — see Add/Remove Files from Project for more information.

The syntax of the `#include` directive has two formats:

```
#include <header_name>
#include "header_name"
```

The preprocessor removes the `#include` line and replaces it with the entire text of a header file at that point in the source code. The placement of `#include` can therefore influence the scope and duration of any identifiers in the included file.

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

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

1. the mikroC for dsPIC30/33 and PIC24 installation folder › “include” folder
2. user's custom search paths

The "`header_name`" version specifies a user-supplied include file; the mikroC for dsPIC30/33 and PIC24 will look for the header file in the following locations, in this particular order:

1. the project folder (folder which contains the project file `.ppc`)
2. the mikroC for dsPIC30/33 and PIC24 installation folder › “include” folder
3. user's custom search paths .

Explicit Path

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

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

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

```
#include macro_identifier
```

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

Preprocessor Operators

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

Operator #

In C preprocessor, a character sequence enclosed by quotes is considered a token and its content is not analyzed. This means that macro names within quotes are not expanded.

If you need an actual argument (the exact sequence of characters within quotes) as a result of preprocessing, use the `#` operator in macro body. It can be placed in front of a formal macro argument in definition in order to convert the actual argument to a string after replacement.

For example, let's have macro `LCD_PRINT` for printing variable name and value on LCD:

```
#define LCD_PRINT(val) Lcd_Custom_Out_Cp(#val ": "); \
                       Lcd_Custom_Out_Cp(IntToStr(val));
```

(note the backslash as a line-continuation symbol)

Now, the following code,

```
LCD_PRINT(temp)
```

will be preprocessed to this:

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

Operator ##

Operator `##` is used for token pasting. Two tokens can be pasted(merged) together by placing `##` in between them (plus optional whitespace on either side). The pre-processor removes whitespace and `##`, combining the separate tokens into one new token. This is commonly used for constructing identifiers.

For example, see the definition of macro SPLICE for pasting two tokens into one identifier:

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

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

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

Conditional Compilation

Conditional compilation directives are typically used to make source programs easy to change and easy to compile in different execution environments. The mikroC for dsPIC30/33 and PIC24 supports conditional compilation by replacing the appropriate source-code lines with a blank line.

All conditional compilation directives must be completed in the source or include file in which they have begun.

Directives #if, #elif, #else, and #endif

The conditional directives #if, #elif, #else, and #endif work very similar to the common C conditional statements. If the expression you write after #if has a nonzero value, the line group immediately following the #if directive is retained in the translation unit.

Syntax is:

```
#if constant_expression_1
<section_1>

[ #elif constant_expression_2
<section_2>]
   ...
[ #elif constant_expression_n
<section_n>]

[ #else
<final_section>]

#endif
```

Each #if directive in a source file must be matched by a closing #endif directive. Any number of #elif directives can appear between #if and #endif directives, but at most one #else directive is allowed. The #else directive, if present, must be the last directive before #endif.

sections can be any program text that has meaning to compiler or preprocessor. The preprocessor selects a single section by evaluating constant_expression following each #if or #elif directive until it finds a true (nonzero) constant expression. The constant expressions are subject to macro expansion.

If all occurrences of constant-expression are false, or if no #elif directives appear, the preprocessor selects the text block after the #else clause. If the #else clause is omitted and all instances of constant_expression in the #if block are false, no section is selected for further processing.

Any processed *section* can contain further conditional clauses, nested to any depth. Each nested #else, #elif, or #endif directive belongs to the closest preceding #if directive.

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

Directives #ifdef and #ifndef

The #ifdef and #ifndef directives can be used anywhere #if can be used and they can test whether an identifier is currently defined or not. The line

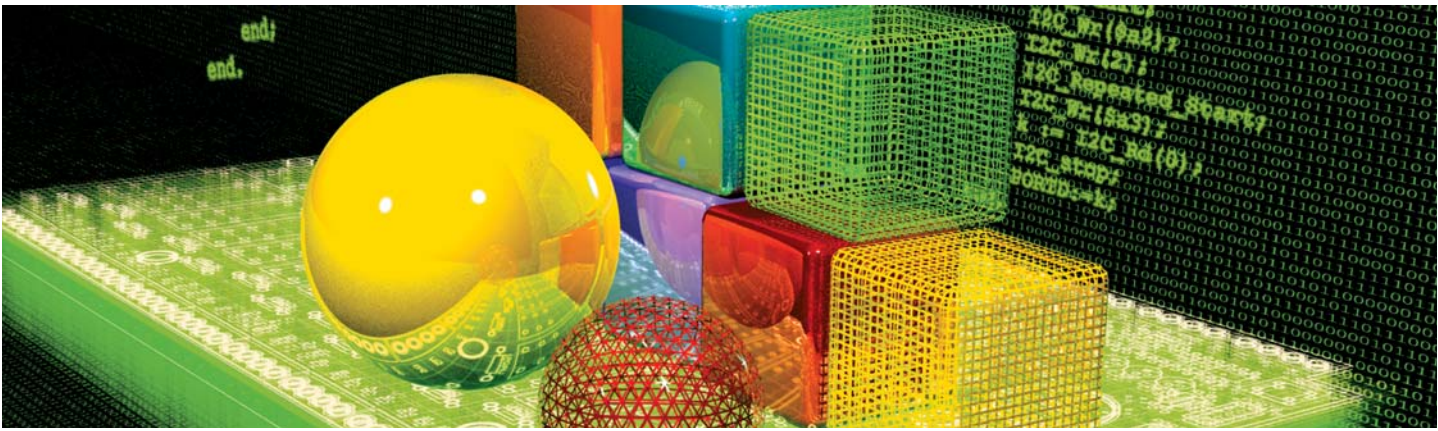
```
#ifdef identifier
```

has exactly the same effect as #if 1 if identifier is currently defined, and the same effect as #if 0 if identifier is currently undefined. The other directive, #ifndef, tests true for the "not-defined" condition, producing the opposite results.

The syntax thereafter follows that of #if, #elif, #else, and #endif.

An identifier defined as NULL is considered to be defined.

CHAPTER 4



mikroC for dsPIC30/33 and PIC24 Libraries

mikroC for dsPIC provides a number of built-in and library routines which help you develop your application faster and easier. ADC Library, Advanced SPI Ethernet 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 Library, LCD8 Custom 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,ANSI C Ctype Library, ANSI C Math Library, ANSI C Stdlib Library, ANSI C String Library, Miscellaneous Libraries,Conversions Library, Setjmp Library, Sprint Library, Time Library, Trigonometry Library, Util Library and Built-in Routines are included along with practical, ready-to-use code examples.

BUILT-IN ROUTINES

The mikroC for dsPIC30/33 and PIC24 compiler provides a set of useful built-in utility functions.

The `Lob`, `Hib`, `Higherb`, `Highestb`, `Lo` and `Hi` routines are implemented as macros. If you want to use these functions you must include `built_in.h` header file (located in the `inlclude` folder of the compiler) into your project.

The `Delay_us` and `Delay_ms` routines are implemented as “inline”; i.e. code is generated in the place of a call, so the call doesn’t count against the nested call limit.

The `Vdelay_ms`, `Delay_Cyc` and `Get_Fosc_kHz` are actual C routines. Their sources can be found in `Delays.c` file located in the `uses` folder of the compiler.

```
Lob
Hib
Higherb
Highestb

Lo
Hi

Delay_us
Delay_ms
Vdelay_ms
Delay_Cyc

Get_Fosc_kHz
```

Lob

Prototype	`unsigned short Lob(unsigned long number);`
Returns	Low byte of number, bits 7..0.
Description	The function returns low byte of `number`. The function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register.This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit. Parameters : `number` - input number
Requires	Nothing.

Example	`unsigned char tmp`
	`unsigned long d = 0xB1AC30F4;`
	`...`
	`tmp = Lob(d);  // tmp equals to 0xF4`

Hib

Prototype	`unsigned short Hib(unsigned long number);`
Returns	High byte of number, bits 15..8.
Description	The function returns high byte of `number`. The function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit. Parameters : `number` - input number
Requires	Nothing.

Example	`unsigned char tmp`
	`unsigned long d = 0xB1AC30F4;`
	`...`
	`tmp = Hib(d);  // tmp equals to 0x30`

Higherb

Prototype	`unsigned short Higherb(unsigned long number);`
Returns	Higher byte of number, bits 23..16.
Description	The function returns higher byte of `number`. The function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit. Parameters : `number`: input number
Requires	Nothing.

Example	`unsigned char tmp`
	`unsigned long d = 0xB1AC30F4;`
	`...`
	`tmp = Higherb(d);  // tmp equals to 0xAC`

Highestb

Prototype	`unsigned short Highestb(unsigned long number);`
Returns	Highest byte of `number`, bits 31..24.
Description	The function returns highest byte of `number`. The function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit. Parameters : `number` - input number.
Requires	Nothing.

Example	`unsigned char tmp`
	`unsigned long d = 0xB1AC30F4;`
	`...`
	`tmp = Highestb(d);  // tmp equals to 0xB1`

Hi

Prototype	`unsigned int Hi(unsigned long number);`
Returns	High word of number, bits `31..16`.
Description	The function returns high word of `number`. The function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit. Parameters : `number`: input number.
Requires	Nothing.

Example	`unsigned tmp`
	`unsigned long d = 0xB1AC30F4;`
	`...`
	`tmp = Hi(d);  // tmp equals to 0xB1AC`

Lo

Prototype	`unsigned int Lo(unsigned long number);`
Returns	Low word of number, bits 15..0.
Description	The function returns low word of `number`. The function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit. Parameters : `number`: input number.
Requires	Nothing.

Example	`unsigned tmp`
	`unsigned long d = 0xB1AC30F4;`
	`...`
	`tmp = Lo(d);  // tmp equals to 0x30F4`

Delay_us

Prototype	`void Delay_us(const unsigned long time_in_us);`
Description	Creates a software delay in duration of `time_in_us` microseconds. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit. 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	`void Delay_ms(const unsigned int time_in_ms);`
Description	Creates a software delay in duration of `time_in_ms` microseconds. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit. 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.
Example	`Delay_ms(1000);  /* One second pause */`

Vdelay_ms

Prototype	`void Vdelay_ms(unsigned Time_ms);`
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` is a library function rather than a built-in routine; it is presented in this topic for the sake of convenience.

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

Delay_Cyc

Prototype	`void Delay_Cyc(unsigned int x, unsigned int y);`
Description	Creates a delay based on MCU clock. Delay lasts for `x*16384 + y` MCU clock cycles. Parameters : `x`: NumberOfCycles divided by 16384 `y`: remaining 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 */`

Get_Fosc_kHz

Prototype	`unsigned long Get_Fosc_kHz(void);`
Returns	Device clock in KHz.
Description	The 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	`unsigned long clk;`
	`...`
	`clk = Clock_Khz();`

LIBRARY ROUTINES

mikroC for dsPIC30/33 and PIC24 provides a set of libraries which simplifies the initialization and use of dsPIC30/33 and PIC24 MCU and its modules.
Currently available libraries are:

- ADC Library
- Advanced SPI Ethernet 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 Library
- LCD8 Custom 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
- ANSI C Ctype Library
- ANSI C Math Library
- ANSI C Stdlib Library
- ANSI C String Library
- Conversions Library
- Setjmp Library
- Sprint Library
- Time Library
- Trigonometry 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 you a comfortable work with the module.

Library Routines

dsPIC30 and PIC24FJ Functions

```
ADC_Read
```

dsPIC33F and PIC24HJ Functions

```
Adc1_Read
Adc2_Read
Adc_Read
```

dsPIC30 and PIC24FJ Functions

Adc_Read

Prototype	`unsigned Adc_Read(unsigned channel);`
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 to enable analog function of the chosen pin.
Requires	The dsPIC30F and PIC24FJ MCU with built-in ADC module. A 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	`unsigned ch_1;`
	`...`
	`ch_1 = Adc_Read(1);  /* read analog value from channel 1 */`

dsPIC33F and PIC24HJ Functions

Adc1_Read

Prototype	`unsigned Adc1_Read(unsigned channel, unsigned AdcMode)`
Returns	10-bit or 12-bit (MCU dependent) unsigned value from the specified `channel`.
Description	The function reads the specified analog channel input. This library works with 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 to enable analog function of the chosen pin.
Requires	The dsPIC33FJ and PIC24HJ MCU with built-in ADC1 module. A 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	`unsigned ch_1;`
	`...`
	`ch_1 = Adc1_Read(1,ADC_10bit);`
	`/* read analog value from channel 1 in 10-bit resolution */`

Adc2_Read

Prototype	`unsigned Adc2_Read(unsigned channel, unsigned AdcMode)`
Returns	10-bit or 12-bit (MCU dependent) 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 chosen pin.
Requires	The dsPIC33FJ and PIC24HJ MCU with built-in ADC2 module. A 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	`unsigned ch_1;`
	`...`
	`ch_1 = Adc2_Read(1,ADC_12bit);`
	`/* read analog value from channel 1 in 12-bit resolution */`

Adc_Read

Prototype	`unsigned Adc_Read(unsigned channel, unsigned AdcMode)`
Returns	10-bit or 12-bit (MCU dependent) unsigned value from the specified `channel`.
Description	This function calls Adc1_Read. It is kept to maintain backwards compatibility with previous versions of the mikroC for dsPIC30/33 and PIC24.
Requires	The dsPIC33FJ and PIC24HJ MCU with built-in ADC1 module. A 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	`unsigned ch_1;`
	`...`
	`ch_1 = Adc_Read(1, ADC_12bit);`
	`// read analog value from ADC1 module channel 1 in 12-bit res.`

Library Example

dsPIC30 and PIC24FJ Functions

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

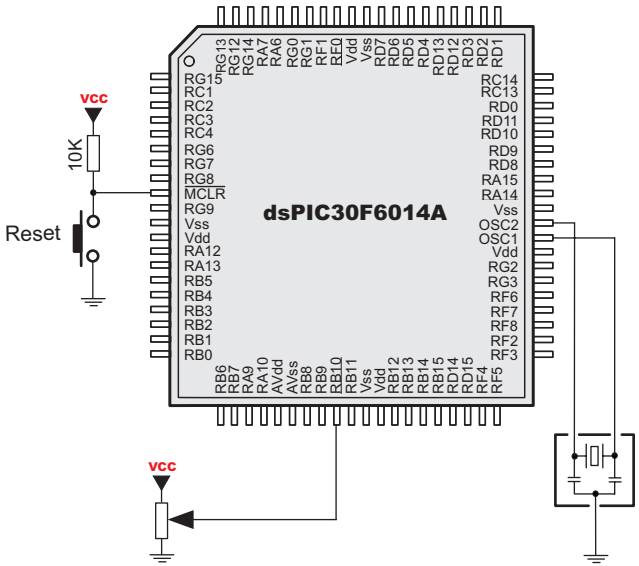
```
unsigned adcRes;
char txt[ 6] ;

void Uart1_Write_Text(char *txt_to_wr) {
  while (*txt_to_wr)
  Uart1_Write_Char(*(txt_to_wr++));
}

void main() {

  TRISBbits.TRISB10 = 1; // set pin as input - needed for ADC to work
  Uart1_Init(9600);
  while (1) {
    adcRes = Adc_Read(10);
    WordToStr(adcRes, txt);
    Uart1_Write_Text(txt);
    Delay_ms(50);
  }
} //~!
```

dsPIC33F and PIC24HJ Functions

In this simple example, the analog input value is read from the channel 1 in 12-bit resolution and sent to USART.

```
unsigned adcRes;
char txt[6];

void Uart1_Write_Text(char *txt_to_wr) {
  while (*txt_to_wr)
  Uart1_Write_Char(*(txt_to_wr++));
}

void main() {

  TRISBbits.TRISB1 = 1; // set pin as input - needed for ADC to work

  Uart1_Init(9600);

  while (1) {
    adcRes = Adc1_Read(1, ADC_12bit);
    WordToStr(adcRes, txt);
    Uart1_Write_Text(txt);
    Delay_ms(50);
  }
} //~!
}
```

Hardware Connection



Advanced SPI Ethernet Library

This library is designed to simplify handling of the underlying hardware (ENC28J60). However, certain level of knowledge about the Ethernet and Ethernet-based protocols (ARP, IP, TCP/IP, UDP/IP, ICMP/IP) is expected from the user. The Ethernet is a high–speed and versatile protocol, but it is not a simple one. Once you get used to it, however, you will make your favorite 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 of the IEEE 802.3 specifications. It incorporates a number of packet filtering schemes to limit incoming packets. It also provides an internal DMA module for fast data throughput and hardware assisted IP checksum calculations. Communication with the host controller is implemented via two interrupt pins and the SPI, with data rates of up to 10 Mb/s. Two 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
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	`void EthSetMACAddr(char MACByte1, char MACByte2, char MACByte3, char MACByte4, char MACByte5, char MACByte6);`
Returns	Nothing.
Description	This routine sets MAC address.
Requires	Nothing.
Example	`// Sets MAC address` `EthSetMACAddr(00,02,04,06,08,10);`

EthSetIPAddr

Prototype	`void EthSetIPAddr(char IpByte1, char IpByte2, char IpByte3, char IpByte4);`
Description	This routine sets IP address.
Requires	Nothing.
Example	`// Sets IP address` `EthSetIPAddr(192,168,20,1);`

EthSetIPMask

Prototype	`void EthSetIPMask(char IPMaskByte1, char IPMaskByte2, char IPMaskByte3, char IPMaskByte4);`
Description	This routine sets IP address mask.
Requires	Nothing.
Example	`// Sets address mask` `EthSetIPMask(255,255,255,0);`

EthSetGateWayAddr

Prototype	`void EthSetGateWayAddr(char GateWayByte1, char GateWayByte2, char GateWayByte3, char GateWayByte4);`
Description	This routine sets Gateway IP address.
Requires	Nothing.
Example	`// Sets Gateway IP address` `EthSetGateWayAddr(192,168,20,1);`

EthInit

Prototype	`void EthInit(unsigned int * RstPort, unsigned int RstPin, unsigned int* CSPort, unsigned int CSPin);`
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	`void MACInit(unsigned int * RstPort, unsigned int RstPin, unsigned int* CSPort, unsigned int CSPin);`
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	As specified for the entire library (MAC.c).
Example	`// Initialize MAC Module` `MACInit(PORTF,0,PORTF,1);`

MACIsTxReady

Prototype	`char MACIsTxReady();`
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	As specified for the entire library (MAC.c).

Example	`// Check MAC transmit readiness...`
	`if ( MACIsTxReady() )`
	`{`
	`// Transmit buffer is empty, transmit a message.`
	`...`

MACGetHeader

Prototype	`char MACGetHeader(MAC_ADDR *remote, unsigned char *type);`
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	As specified for the entire library (MAC.c).

Example	`// Get possible data packet info.`
	`if ( MACGetHeader(&RemoteNodeMAC, &PacketType) )`
	`{`
	`// A packet is received, process it.`
	`...`
	`// Once done, discard it.`
	`MACDiscardRx();`
	`...`

MACGet

Prototype	`unsigned char MACGet();`
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) )`
	`{`
	`// A packet is received, process it.`
	`data = MACGet();`
	`...`
	`// When done, discard it.`
	`MACDiscardRx();`
	`...`

MACGetArray

Prototype	`unsigned int MACGetArray(unsigned char *val, unsigned int len);`
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) )`
	`{`
	`// A packet is received, process it.`
	`actualCount = MACGetArray(data, count);`
	`...`

MACDiscardRx

Prototype	`void 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) )`
	`{`
	`// A packet is received, process it.`
	`actualCount = MACGetArray(data, count);`
	`...`
	`// Done processing it. Discard it.`
	`MACDiscardRx();`
	`...`

MACPutHeader

Prototype	`void MACPutHeader(MAC_ADDR *remote, unsigned char type, unsigned int dataLen);`
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() )`
	`{`
	`// Assemble IP packet with total IP packet size of 100 bytes`
	`// including IP header.`
	`MACPutHeader(&RemoteNodeMAC, MAC_IP, 100);`
	`...`

MACPut

Prototype	`void MACPut(unsigned char val);`
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() )`
	`{`
	`// 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(0x55);`
	`...`

MACPutArray

Prototype	`void MACPutArray(unsigned char *val, unsigned int len);`
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() )`
	`{`
	`// 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(0x55);`
	`MACPutArray(data, count);`
	`...`

MACFlush

Prototype	`void MACFlush();`
Description	This function marks active transmit buffer as ready for transmission.
Requires	MACPutHeader or MACSetTxBuffer must have been called.

Example	`// Check to see if at least one transmit buffer is empty`
	`if ( MACIsTxReady() )`
	`{`
	`// 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(0x55);`
	`MACPutArray(data, count);`
	`...`
	`// Now transmit it.`
	`MACFlush();`
	`...`

MACDiscardTx

Prototype	`void MACDiscardTx(BUFFER 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() )`
	`{`
	`// 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);`
	`...`

MACSetRxBuffer

Prototype	`void MACSetRxBuffer(unsigned int offset);`
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) )`
	`{`
	`// A packet is received, process it.`
	`actualCount = MACGetArray(data, count);`
	`...`
	`// Fetch data beginning at offset 20`
	`MACSetRxBuffer(20);`
	`data = MACGet();`
	`...`

MACSetTxBuffer

Prototype	`void MACSetTxBuffer(BUFFER buffer, unsigned int offset);`
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() )`
	`{`
	`// 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();`
	`...`

MACReserveTxBuffer

Prototype	`void MACReserveTxBuffer(BUFFER buffer);`
Description	This function reserves a given transmit buffer and marks it as unavailable. This function is useful for TCP layer where a message would 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 MACGetTxBuffer function.
Requires	Nothing.

Example	`// Check to see if at least one transmit buffer is empty`
	`if ( MACIsTxReady() )`
	`{`
	`// 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();`
	`...`

MACGetFreeRxSize

Prototype	`unsigned int MACGetFreeRxSize();`
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	`void MACSetDuplex(DUPLEX DuplexState);`
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	`char ARPGet(NODE_INFO *remote, unsigned char *opCode);`
Returns	(!=0) - If a valid ARP packet that was addressed to local host was fetched; remote and opCode contain valid values. (==0) - Either unknown ARP code was fetched or this packet was not addressed to 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) )`
	`{`
	`// If this is an ARP packet, fetch it.`
	`If ( PacketType == MAC_ARP )`
	`{`
	`// This is ARP packet.`
	`ARPGet(&RemoteNode, &ARPCode);`
	`...`

ARPPut

Prototype	`void ARPPut(NODE_INFO *remote, unsigned char opCode);`
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() )`
	`{`
	`// Transmit it`
	`ARPPut(&RemoteNode, ARP_REQUEST);`
	`...`

ARPInit

Prototype	`void 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	`char IPIsTxReady();`
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() )`
	`{`
	`// Assemble IP packet.`
	`IPPutHeader(&Remote, MAC_TCP, IPPacketLen);`
	`...`

IPSetTxBuffer

Prototype	`void IPSetTxBuffer(BUFFER buffer, unsigned int offset);`
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() )`
	`{`
	`// 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	`unsigned int IPPutHeader(NODE_INFO *remote, char protocol, unsigned int len);`
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() )`
	`{`
	`// 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	`void IPPutArray(char *buffer, unsigned int len);`
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() )`
	`{`
	`// 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	`char IPGetHeader(IP_ADDR *localIP, NODE_INFO *remote, char *protocol, unsigned int *len);`
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) )`
	`{`
	`// Check what kind of protocol it is`
	`if ( PacketType == MAC_IP )`
	`{`
	`// This is IP packet. Fetch it.`
	`IPGetHeader(&Local, &Remote, &IPProtocol, &IPLen);`
	`// Process this IP packet.`
	`...`
	`// When done processing this packet, discard it`
	`MACDiscardRx();`
	`}`
	`else`
	`{`
	`// This is not an IP packet. Handle it`
	`...`

IPGetArray

Prototype	`unsigned int IPGetArray(char *val, unsigned int len);`
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) )`
	`{`
	`// Check what kind of protocol it is`
	`if ( PacketType == MAC_IP )`
	`{`
	`// 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();`
	`}`
	`else`
	`{`
	`// This is not an IP packet. Handle it`
	`...`

IPSetRxBuffer

Prototype	`void IPSetRxBuffer(unsigned int offset);`
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) )`
	`{`
	`// Check what kind of protocol it is`
	`if ( PacketType == MAC_IP )`
	`{`
	`// 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();`
	`}`
	`else`
	`{`
	`// This is not an IP packet. Handle it`
	`...`

ICMPIsTxReady

Prototype	`char ICMPIsTxReady();`
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() )`
	`{`
	`// Transmit ICMP packet.`
	`...`

ICMPPut

Prototype	`void ICMPPut(NODE_INFO *remote, ICMP_CODE code, char *data, char len, unsigned int id, unsigned int seq);`
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() )`
	`{`
	`// Transmit ICMP packet.`
	`ICMPPut(&RemoteNode, ICMP_ECHO_REPLY, data, datalen, id, seq);`
	`// Done. ICMP is put into transmit queue.`
	`...`

ICMPGet

Prototype	`void ICMPGet(NODE_INFO *remote, ICMP_CODE *code, char *data, char *len, unsigned int *id, unsigned int *seq);`
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) )`
	`{`
	`// Check what kind of protocol it is`
	`if ( IPProtocol == IP_PROT_ICMP )`
	`{`
	`// This is ICMPP packet. Fetch it.`
	`ICMPGet(&ICMPCode, data, &dataLen, &id, &seq);`
	`// Process this ICMP packet.`
	`...`
	`// When done processing this packet, discard it`
	`MACDiscardRx();`
	`}`
	`else`
	`{`
	`// This is not an ICMP packet. Handle it`
	`...`

TCPInit

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

TCPListen

Prototype	`TCP_SOCKET TCPListen(TCP_PORT port);`
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	`...`
	`switch(smState)`
	`{`
	`case SM_LISTEN:`
	`// Listen for HTTP requests.`
	`httpSocket = TCPListen(80);`
	`If ( httpSocket == INVALID_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`...`
	`}`
	`else`
	`smState = SM_LISTEN_WAIT;`
	`return;`
	`case SM_LISTEN_WAIT:`
	`// Wait for connection...`
	`...`

TCPConnect

Prototype	`TCP_SOCKET TCPConnect(NODE_INFO *remote, TCP_PORT port);`
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	`...`
	`switch(smState)`
	`{`
	`case SM_CONNECT:`
	`// Connect to a remote FTP server.`
	`ftpSocket = TCPConnect(&RemoteNode, 21);`
	`If ( ftpSocket == INVALID_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`smState = SM_CONNECT_WAIT;`
	`return;`
	`case SM_CONNECT_WAIT:`
	`// Wait for connection...`
	`...`

TCPIsConnected

Prototype	`char TCPIsConnected(TCP_SOCKET socket);`
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	`switch(smState)`
	`{`
	`case SM_CONNECT:`
	`// Connect to a remote FTP server.`
	`ftpSocket = TCPConnect(&RemoteNode, 21);`
	`If ( ftpSocket == INVALID_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`smState = SM_CONNECT_WAIT;`
	`return;`
	`case SM_CONNECT_WAIT:`
	`// Wait for connection...`
	`if ( TCPIsConnected(ftpSocket) )`
	`smState = SM_CONNECTED;`
	`return;`
	`...`

TCPDisconnect

Prototype	`void TCPDisconnect(TCP_SOCKET socket);`
Returns	Nothing.
Description	This function requests remote host to disconnect.
Requires	Nothing.

Example	`switch(smState)`
	`{`
	`case SM_CONNECT:`
	`// Connect to a remote FTP server.`
	`ftpSocket = TCPConnect(&RemoteNode, 21);`
	`If ( ftpSocket == INVALID_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`smState = SM_CONNECT_WAIT;`
	`return;`
	`case SM_CONNECT_WAIT:`
	`// Wait for connection...`
	`if ( TCPIsConnected(ftpSocket) )`
	`smState = SM_CONNECTED;`
	`return;`
	`case SM_CONNECTED:`
	`// Send data`
	`...`
	`// Disconnect`
	`TCPDisconnect(ftpSocket);`
	`...`

TCPIsPutReady

Prototype	`char TCPIsPutReady(TCP_SOCKET socket);`
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 its transmit buffer is empty. `socket` - Socket identifier that needs to be checked.
Requires	Nothing.

Example	`...`
	`switch(smState)`
	`{`
	`case SM_CONNECT:`
	`// Connect to a remote FTP server.`
	`ftpSocket = TCPConnect(&RemoteNode, 21);`
	`If ( ftpSocket == INVALID_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`smState = SM_CONNECT_WAIT;`
	`return;`
	`case SM_CONNECT_WAIT:`
	`// Wait for connection...`
	`if ( TCPIsConnected(ftpSocket) )`
	`smState = SM_CONNECTED;`
	`return;`
	`case SM_CONNECTED:`
	`// Send data`
	`if ( TCPIsPutReady(ftpSocket) )`
	`{`
	`// Send data`
	`...`

TCPPut

Prototype	`char TCPPut(TCP_SOCKET socket, char 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. `byte` - Data byte to be loaded.
Requires	TCPIsPutReady == TRUE

Example	`...`
	`switch(smState)`
	`{`
	`case SM_CONNECT:`
	`// Connect to a remote FTP server.`
	`ftpSocket = TCPConnect(&RemoteNode, 21);`
	`If ( ftpSocket == INVALID_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`smState = SM_CONNECT_WAIT;`
	`return;`
	`case SM_CONNECT_WAIT:`
	`// Wait for connection...`
	`if ( TCPIsConnected(ftpSocket) )`
	`smState = SM_CONNECTED;`
	`return;`
	`case SM_CONNECTED:`
	`// Send data`
	`if ( TCPIsPutReady(ftpSocket) )`
	`{`
	`// Send data`
	`TCPPut(ftpSocket, dataByte);`
	`...`

TCPFlush

Prototype	`void TCPFlush(TCP_SOCKET 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	```...
	switch(smState)
	{
	case SM_CONNECT:
	// Connect to a remote FTP server.
	ftpSocket = TCPConnect(&RemoteNode, 21);
	If (ftpSocket == INVALID_SOCKET)
	{
	// Socket is not available
	// Return error.
	}
	else
	smState = SM_CONNECT_WAIT;
	return;
	case SM_CONNECT_WAIT:
	// Wait for connection...
	if (TCPIsConnected(ftpSocket))
	smState = SM_CONNECTED;
	return;
	case SM_CONNECTED:
	// Send data
	if (TCPIsPutReady(ftpSocket))
	{
	// Send data
	TCPPut(ftpSocket, dataByte);
	...
	// Now transmit it.
	TCPFlush(ftpSocket);
	...```

TCPIsGetReady

Prototype	`char TCPIsGetReady(TCP_SOCKET socket);`
Returns	Nothing.
Description	(`!=0`) - If given socket contains receive data. (`==0`) - If given socket does not contain any data.
Requires	This function determines if the given socket contains receive data. `socket` - Socket identifier that needs to be transmitted.

Example	`...`
	`switch(smState)`
	`{`
	`case SM_LISTEN:`
	`// Listen to HTTP socket`
	`httpSocket = TCPListen(&RemoteNode, 80);`
	`If ( httpSocket == INVALID_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`smState = SM_LISTEN_WAIT;`
	`return;`
	`case SM_LISTEN_WAIT:`
	`// Wait for connection...`
	`if ( TCPIsConnected(httpSocket) )`
	`smState = SM_CONNECTED;`
	`return;`
	`case SM_CONNECTED:`
	`// Fetch data`
	`if ( TCPIsGetReady(httpSocket) )`
	`{`
	`// Fetch data`
	`...`

TCPGet

Prototype	`char TCPGet(TCP_SOCKET socket, char *byte);`
Returns	(`!=0`) - If a byte was read. (`==0`) - If no byte was read.
Description	This function fetches one data byte from a given socket receive buffer. `socket` - Socket identifier that needs to be fetched. `byte` - Data byte that was read.
Requires	TCPIsGetReady == TRUE

Example	`...`
	`switch(smState)`
	`{`
	`case SM_LISTEN:`
	`// Listen to HTTP socket`
	`httpSocket = TCPListen(&RemoteNode, 80);`
	`If ( httpSocket == INVALID_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`smState = SM_LISTEN_WAIT;`
	`return;`
	`case SM_LISTEN_WAIT:`
	`// Wait for connection...`
	`if ( TCPIsConnected(httpSocket) )`
	`smState = SM_CONNECTED;`
	`return;`
	`case SM_CONNECTED:`
	`// Fetch data`
	`if ( TCPIsGetReady(httpSocket) )`
	`{`
	`// Fetch data`
	`TCPGet(httpSocket, &dataByte);`
	`...`

TCPGetArray

Prototype	`unsigned int TCPGetArray(TCP_SOCKET socket, char *byte, unsigned int count);`
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. `byte` - Data array that was read. `count` - Total number of bytes to read.
Requires	TCPIsGetReady == TRUE

Example	`...`
	`switch(smState)`
	`{`
	`case SM_LISTEN:`
	`// Listen to HTTP socket`
	`httpSocket = TCPListen(&RemoteNode, 80);`
	`If ( httpSocket == INVALID_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`smState = SM_LISTEN_WAIT;`
	`return;`
	`case SM_LISTEN_WAIT:`
	`// Wait for connection...`
	`if ( TCPIsConnected(httpSocket) )`
	`smState = SM_CONNECTED;`
	`return;`
	`case SM_CONNECTED:`
	`// Fetch data`
	`if ( TCPIsGetReady(httpSocket) )`
	`{`
	`// Fetch 20 bytes of data`
	`TCPGetArray(httpSocket, buffer, 20);`
	`...`

TCPDiscard

Prototype	`char TCPDiscard(TCP_SOCKET socket);`
Returns	(!=0) - If receive buffer for given was successfully discarded. (==0) - If receive buffer for given buffer 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	```...
	switch(smState)
	{
	case SM_LISTEN:
	// Listen to HTTP socket
	httpSocket = TCPListen(&RemoteNode, 80);
	If (httpSocket == INVALID_SOCKET)
	{
	// Socket is not available
	// Return error.
	}
	else
	smState = SM_LISTEN_WAIT;
	return;
	case SM_LISTEN_WAIT:
	// Wait for connection...
	if (TCPIsConnected(httpSocket))
	smState = SM_CONNECTED;
	return;
	case SM_CONNECTED:
	// Fetch data
	if (TCPIsGetReady(httpSocket))
	{
	// Fetch 20 bytes of data
	TCPGetArray(httpSocket, buffer, 20);
	// Process data.
	...
	// Release the buffer.
	TCPDiscard(httpSocket);
	...```

TCPProcess

Prototype	`char TCPProcess(NODE_INFO *remote, unsigned int len);`
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	`...`
	`switch(smState)`
	`{`
	`case SM_STACK_IDLE:`
	`if ( MACGetHeader(&RemoveMAC, &MACFrameType) )`
	`{`
	`if ( MACFrameType == MAC_IP )`
	`smState = SM_STACK_IP;`
	`...`
	`return;`
	`case SM_STACK_IP:`
	`if ( IPGetHeader(&RemoteNode, &IPFrameType, &IPDataCount) )`
	`{`
	`if ( IPFrameType == IP_PROT_TCP )`
	`smState = SM_STACK_TCP;`
	`...`
	`return;`
	`case SM_STACK_TCP:`
	`if ( TCPProcess(&RemoteNode, IPDataCount) )`
	`smState = SM_STACK_IDLE;`
	`return;`
	`...`

TCPTick

Prototype	`void 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	`void UDPInit();`
Description	This function initializes the UDP module and prepares it for multiple UDP connections.
Requires	Nothing.
Example	`// Initialize UDP` `UDPInit();`

UDPOpen

Prototype	`UDP_SOCKET UDPOpen(UDP_PORT localPort, NODE_INFO *remoteNode, TCP_PORT remotePort);`
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.

Example	`...`
	`switch(smState)`
	`{`
	`case SM_OPEN:`
	`// Talk to a remote DHCP server.`
	`DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);`
	`If ( DHCPSocket == INVALID_UDP_SOCKET )`
	`{`
	`// Socket is not available. Return error.`
	`}`
	`else // Broadcast DHCP Broadcast message.`
	`break;`
	`...`

UDPClose

Prototype	`void UDPClose(UDP_SOCKET socket);`
Returns	Nothing.
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	`...`
	`switch(smState)`
	`{`
	`case SM_OPEN:`
	`// Talk to a remote DHCP server.`
	`DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);`
	`If ( DHCPSocket == INVALID_UDP_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`// Send DHCP request...`
	`...`
	`// Close the socket`
	`UDPClose(DHCPSocket);`
	`break;`
	`...`

UDPIsPutReady

Prototype	`char UDPIsPutReady(UDP_SOCKET socket);`
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	`...`
	`switch(smState)`
	`{`
	`case SM_OPEN:`
	`// Talk to a remote DHCP server.`
	`DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);`
	`If ( DHCPSocket == INVALID_UDP_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`// Broadcast DHCP Broadcast message.`
	`smState = SM_BROADCAST;`
	`break;`
	`case SM_BROADCAST:`
	`if ( UDPIsPutReady(DHCPSocket) )`
	`{`
	`// Socket is ready to transmit. Transmit the data...`
	`...`
	`}`
	`break;`
	`...`

UDPPut

Prototype	`char UDPPut(unsigned char 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. `byte` - Data byte to be loaded.
Requires	UDPIsPutReady == TRUE
Example	...

```
...
switch(smState)
{
case SM_OPEN:
// Talk to a remote DHCP server.
DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
If ( DHCPSocket == INVALID_UDP_SOCKET )
{
// Socket is not available
// Return error.
}
else
// Broadcast DHCP Broadcast message.
smState = SM_BROADCAST;
break;
case SM_BROADCAST:
if ( UDPIsPutReady(DHCPSocket) )
{
// 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(0x55);
...
}
break;
...
```

UDPFlush

Prototype	`void 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	...

```
...
switch(smState)
{
case SM_OPEN:
// Talk to a remote DHCP server.
DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
If ( DHCPSocket == INVALID_UDP_SOCKET )
{
// Socket is not available
// Return error.
}
else
// Broadcast DHCP Broadcast message.
smState = SM_BROADCAST;
break;
case SM_BROADCAST:
if ( UDPIsPutReady(DHCPSocket) )
{
// 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(0x55);
...
// Now transmit it.
UDPFlush();
}
break;
...
```

UDPIsGetReady

Prototype	`char UDPIsGetReady(UDP_SOCKET socket);`
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	`switch(smState)`
	`{`
	`case SM_OPEN:`
	`// Talk to a remote DHCP server.`
	`DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);`
	`If ( DHCPSocket == INVALID_UDP_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`// Wait for response from DHCP server`
	`smState = SM_WAIT_FOR_DATA;`
	`break;`
	`case SM_WAIT_FOR_DATA:`
	`if ( UDPIsGetReady(DHCPSocket) )`
	`{`
	`// Socket does contain some data. Fetch it and process it.`
	`...`
	`}`
	`break;`
	`...`

UDPGet

Prototype	`char UDPGet(unsigned char *byte);`
Returns	(!=0) If a byte was read. (==0) If no byte was read.
Description	This function fetches one data byte from an active socket receive buffer. `byte` - Data byte that was read.
Requires	UDPIsGetReady == TRUE

Example	`...`
	`switch(smState)`
	`{`
	`case SM_OPEN:`
	`// Talk to a remote DHCP server.`
	`DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);`
	`If ( DHCPSocket == INVALID_UDP_SOCKET )`
	`{`
	`// Socket is not available`
	`// Return error.`
	`}`
	`else`
	`// Wait for response from DHCP server`
	`smState = SM_WAIT_FOR_DATA;`
	`break;`
	`case SM_WAIT_FOR_DATA:`
	`if ( UDPIsGetReady(DHCPSocket) )`
	`{`
	`// Socket does contain some data. Fetch it all.`
	`// buffer is a pointer to BYTE.`
	`while( UDPGet(buffer) )`
	`buffer++;`
	`// Process it.`
	`...`
	`// Discard the socket buffer.`
	`...`
	`}`
	`break;`
	`...`

UDPDiscard

Prototype	`char UDPDiscard();`
Returns	Nothing.
Description	This function releases the receive buffer associated with an active socket.
Requires	Nothing.

Example	```...
	switch(smState)
	{
	case SM_OPEN:
	// Talk to a remote DHCP server.
	DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
	If (DHCPSocket == INVALID_UDP_SOCKET)
	{
	// Socket is not available
	// Return error.
	}
	else
	// Wait for response from DHCP server
	smState = SM_WAIT_FOR_DATA;
	break;
	case SM_WAIT_FOR_DATA:
	if (UDPIsGetReady(DHCPSocket))
	{
	// Socket does contain some data. Fetch it all.
	// buffer is a pointer to BYTE.
	while(UDPGet(buffer))
	buffer++;
	// Process it..
	...
	// Discard the socket buffer.
	UDPDiscard();
	}
	break;
	...```

UDPProcess

Prototype	`char UDPProcess(NODE_INFO *remote, unsigned int len);`
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	`...`
	`switch(smState)`
	`{`
	`case SM_STACK_IDLE:`
	`if ( MACGetHeader(&RemoveMAC, &MACFrameType) )`
	`{`
	`if ( MACFrameType == MAC_IP )`
	`smState = SM_STACK_IP;`
	`...`
	`return;`
	`case SM_STACK_IP:`
	`if ( IPGetHeader(&RemoteNode, &IPFrameType, &IPDataCount) )`
	`{`
	`if ( IPFrameType == IP_PROT_UDP )`
	`smState = SM_STACK_UDP;`
	`...`
	`return;`
	`case SM_STACK_UDP:`
	`if ( UDPProcess(&RemoteNode, IPDataCount) )`
	`smState = SM_STACK_IDLE;`
	`return;`
	`...`

UDPWrite

Prototype	`unsigned char UDPWrite(unsigned int UDPSocket, char * UDPBuffer, char NoBytes);`
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	`unsigned char UDPRead(unsigned int UDPSocket, char * UDPBuffer);`
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	`unsigned int UDPOpenSocket(unsigned int LocUDPPort, char RemoteIPByte1, char RemoteIPByte2, char RemoteIPByte3, char RemoteIPByte4, unsigned int RemUDPPort);`
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	`void StackInit(unsigned int * RstPort, unsigned int RstPin, unsigned int* CSPort, unsigned int CSPin);`
Returns	Nothing.
Description	This routine initializes stack and its components.
Requires	Nothing.
Example	`StackInit(PORTF, 0, PORTF, 1); //initialize stack`

StackTask

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

HTTPInit

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

HTTPServer

Prototype	`void HTTPServer(void);`
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 SPI Ethernet Library. dsPIC is assigned an IP address of 192.168.20.25 and will respond to ping if connected to a local area network.

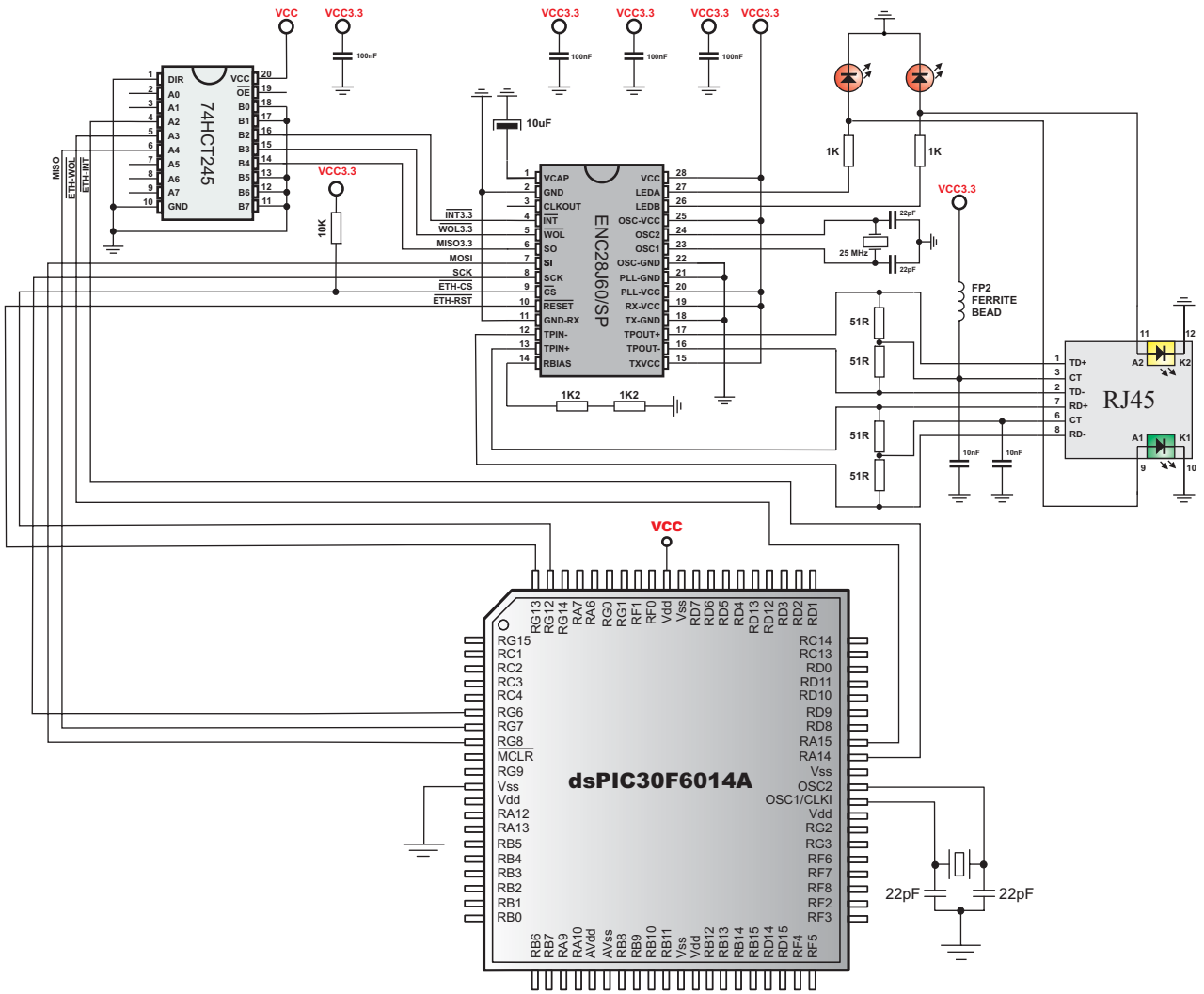
```
#include "SerEth.h"

//#include "Tick.h"
#define BaudRate 9600
#define DefaultUDPPort 10001

void *CopyConst2Ram(void * dest, const void * sors, unsigned char
n)
{
     char *      ramptr;
     const char *    constptr;

     constptr = sors;
     ramptr = dest;
     while(n--)
         *ramptr++ = *constptr++;
     return dest;
}


unsigned int main_socket;
static char Buffer0[ 8] ;
static char Buffer1[ 8] ;

void Interrupt_T1 (void) org 0x001A
{
    TimeOutUpdate();
}

static void ReadAnalogInputs(void)
{
     unsigned int ADCResult;

     ADCResult = Adc_Read(0);
     IntToStr(ADCResult, Buffer0);
     ADCResult = Adc_Read(1);
     IntToStr(ADCResult, Buffer1);
}

#define DISPLAY       (0)

// continues...
```

```
// continued...

#define LED_D2           (0x00)
#define LED_D3           (0x01)
#define ANALOGINPUT_0    (0x02)
#define ANALOGINPUT_1    (0x03)
#define SETLED_D2           (0x04)
#define SETLED_D3           (0x05)


const char HTTP_DEFAULT_PAGE[ ]  = "CENTER.HTM";
const char EXECUTE_PAGE[ ]  = "EXECUTE.CGI";
const char UNKNOWN_PAGE[ ]  = "UNKNOWN.HTM";

void HTTPExecCmd(unsigned char** argv, unsigned char argc)
{
     char command;
     char var;

     command = argv[ 0][ 0]  - '0';
     switch(command)
     {
     case DISPLAY:
          var = argv[ 1][ 0]  - '0';

          switch(var)
          {
          case LED_D2:
               LATDbits.LATD2 ^= 1;
               break;

          case LED_D3:
               LATDbits.LATD3 ^= 1;
               break;
           }

           CopyConst2Ram((void*)argv[ 0] ,
                 (const void*)EXECUTE_PAGE, sizeof(EXECUTE_PAGE));
           break;

     default:
          CopyConst2Ram((void*)argv[ 0] ,
                 (const void*)UNKNOWN_PAGE, sizeof(UNKNOWN_PAGE));
          break;
     }
}

//continues...
```

```
// continued...

unsigned int HTTPGetVar(char var, unsigned int ref, char* val)
{
     switch(var)
     {

     case LED_D2:
          *val = LATDbits.LATD2 ? '1':'0';
          break;

     case LED_D3:
          *val = LATDbits.LATD3 ? '1':'0';
          break;

     case ANALOGINPUT_0:
          *val = Buffer0[(char)ref];
          if ( Buffer0[(char)ref] == '\0' )
               return 0xFFFF;

          (char)ref++;
          return ref;

     case ANALOGINPUT_1:
          *val = Buffer1[(char)ref];
          if ( Buffer1[(char)ref] == '\0' )
               return 0xFFFF;

          (char)ref++;
          return ref;

     case SETLED_D2:
          *val = LATDbits.LATD2 ? '0':'1';
          break;

     case SETLED_D3:
          *val = LATDbits.LATD3 ? '0':'1';
          break;
     }
     return 0xFFFF;
}

char Bbuffer[20] = "                    ";
char i = 0;


// continues..
```

```
// continued...
void main(void)
{
     static unsigned long t = 0;
     ADPCFG = 0xFFFF;
     PORTD = 0;
     TRISD = 0;
     // Set up analog inputs
     TRISBbits.TRISB0 = 1;
     TRISBbits.TRISB1 = 1;
     // Set up the LEDs.
     LATD = 0x00;
     LATDbits.LATD2 = 1;
     LATDbits.LATD3 = 1;
     TRISD = 0x00;
     // Initialize Usart module
     Uart1_Init(19200);
     U1MODEbits.ALTIO = 1;
     Delay_ms(100);
     // Enable interrupts
     INTCON1bits.NSTDIS = 1;
     EthSetIPAddr(192, 168, 20, 60);
     EthSetMACAddr(0x00, 0x02, 0x04, 0x06, 0x08, 0x0A);
     EthSetIPMask(0xFF, 0xFF, 0xFF, 0x00);
     EthSetGateWayAddr(192, 168, 20, 60);
     EthInit( &PORTF, 0, &PORTF, 1);
     // Open UDP communication sockets needed
     main_socket = UDPOpenSocket(DefaultUDPPort, 192, 168, 20, 1,
DefaultUDPPort);
     while(1)
     {
          if ( TickGetDiff(TickGet(), t) >= TICK_SECOND/2 )
          {
               t = TickGet();
               LATDbits.LATD0 ^= 1;
          }
          StackTask();

         // Process incoming UDP packets and send a reply if needed.
         // User specific code.

          if(i = UDPRead(main_socket, Bbuffer)) {
            Bbuffer[ i]  = 0;
            UDPWrite(main_socket, Bbuffer, strlen(Bbuffer));
          }

          HTTPServer();
          ReadAnalogInputs();
    }
}
```

Hardware Connection



CAN LIBRARY

The mikroC 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 grater 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
- CAN1IDToRegs
- RegsToCAN2ID
- CAN2IDToRegs

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

CAN1SetOperationMode

Prototype	`void CAN1SetOperationMode(unsigned int mode, unsigned int WAIT);`
Description	Sets the CAN1 module to requested mode. Parameters : - `mode` :CAN1 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. 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	`unsigned int CAN1GetOperationMode(void);`
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) {`
	`  ...`
	`}`

CAN1Initialize

Prototype	`void CAN1Initialize(unsigned int SJW, unsigned int BRP, unsigned int PHSEG1, unsigned int PHSEG2, unsigned int PROPSEG, unsigned int CAN_CONFIG_FLAGS);`
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 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 CAN_CONFIG_FLAGS value `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to `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 mes`
	`// sage acceptance flags along with the sampling rules`
	`unsigned int can_config_flags;`
	`...`
	`can_config_flags = CAN_CONFIG_SAMPLE_THRICE &`
	`// Form value to be used`
	`CAN_CONFIG_PHSEG2_PRG_ON &`
	`// with CAN1Initialize`
	`CAN_CONFIG_STD_MSG &`
	`CAN_CONFIG_DBL_BUFFER_ON &`
	`CAN_CONFIG_MATCH_MSG_TYPE &`
	`CAN_CONFIG_LINE_FILTER_OFF;`
	`CAN1Initialize(1,3,3,3,1,can_config_flags);`
	`// initialize the CAN1 module`

CAN1SetBaudRate

Prototype	`void CAN1SetBaudRate(unsigned int SJW, unsigned int BRP, unsigned int PHSEG1, unsigned int PHSEG2, unsigned int PROPSEG, unsigned int CAN_CONFIG_FLAGS);`
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. `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to `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`
	`unsigned int can_config_flags;`
	`...`
	`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 &`
	`// Form value to be used`
	`CAN_CONFIG_PHSEG2_PRG_ON &`
	`// with CAN1SetBaudRate`
	`CAN_CONFIG_STD_MSG &`
	`CAN_CONFIG_DBL_BUFFER_ON &`
	`CAN_CONFIG_MATCH_MSG_TYPE &`
	`CAN_CONFIG_LINE_FILTER_OFF;`
	
	`CAN1SetBaudRate(1,3,3,3,1,can_config_flags);`
	`// set the CAN1 module baud rate`

CAN1SetMask

Prototype	`void CAN1SetMask(unsigned int CAN_MASK, long val, unsigned int CAN_CONFIG_FLAGS);`

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 & CAN_CONFIG_STD_MSG,`
	`CAN_CONFIG_MATCH_MSG_TYPE & CAN_CONFIG_XTD_MSG.`
	(see CAN constants)

Requires	The CAN1 routines are supported only by MCUs with the 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 & CAN_CONFIG_XTD_MSG);`

CAN1SetFilter

Prototype	`void CAN1SetFilter(unsigned int CAN_FILTER, long val, unsigned int CAN_CONFIG_FLAGS);`
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	`unsigned int CAN1Read(unsigned long *id, char *data, unsigned int *dataLen, unsigned int *CAN_RX_MSG_FLAGS);`
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 `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.`
	`unsigned int msg_rcvd, rx_flags, data_len;`
	`char data[ 8] ;`
	`unsigned long msg_id;`
	`...`
	`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)) {`
	`  ...`
	`}`

CAN1Write

Prototype	`unsigned int CAN1Write(long id, char *Data, unsigned int DataLen, unsigned int CAN_TX_MSG_FLAGS);`
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`
	`unsigned int tx_flags;`
	`char data[8];`
	`unsigned long msg_id;`
	`...`
	`CAN1SetOperationMode(CAN_MODE_NORMAL,0xFF);`
	`// set NORMAL mode (CAN1 must be in mode in which transmission is`
	`// possible)`
	
	`tx_flags = ECAN_TX_PRIORITY_0 &`
	`ECAN_TX_XTD_FRAME &`
	`ECAN_TX_NO_RTR_FRAME;`
	`// set message flags`
	`CAN1Write(msg_id, data, 1, tx_flags);`

CAN2SetOperationMode

Prototype	`void CAN2SetOperationMode(unsigned int mode, unsigned int WAIT);`
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. 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	`unsigned int CAN2GetOperationMode(void);`
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) {`
	`  ...`
	`}`

CAN2Initialize

Prototype	`void CAN2Initialize(unsigned int SJW, unsigned int BRP, unsigned int PHSEG1, unsigned int PHSEG2, unsigned int PROPSEG, unsigned int CAN_CONFIG_FLAGS);`
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 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 `CAN_CONFIG_FLAGS` value `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to `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 mes`
	`// sage acceptance flags along with the sampling rules`
	`unsigned int can_config_flags;`
	`...`
	`can_config_flags = CAN_CONFIG_SAMPLE_THRICE &`
	`// Form value to be used`
	`CAN_CONFIG_PHSEG2_PRG_ON &`
	`// with CAN2Initialize`
	`CAN_CONFIG_STD_MSG &`
	`CAN_CONFIG_DBL_BUFFER_ON &`
	`CAN_CONFIG_MATCH_MSG_TYPE &`
	`CAN_CONFIG_LINE_FILTER_OFF;`
	
	`CAN2Initialize(1,3,3,3,1,can_config_flags);`
	`// initialize the CAN2 module`

CAN2SetBaudRate

Prototype	`void CAN2SetBaudRate(unsigned int SJW, unsigned int BRP, unsigned int PHSEG1, unsigned int PHSEG2, unsigned int PROPSEG, unsigned int CAN_CONFIG_FLAGS);`
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. `SAM`, `SEG2PHTS` and `WAKFIL` bits are set according to `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`
	`unsigned int can_config_flags;`
	`...`
	`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 &`
	`// Form value to be used`
	`CAN_CONFIG_PHSEG2_PRG_ON &`
	`// with CAN2SetBaudRate`
	`CAN_CONFIG_STD_MSG &`
	`CAN_CONFIG_DBL_BUFFER_ON &`
	`CAN_CONFIG_MATCH_MSG_TYPE &`
	`CAN_CONFIG_LINE_FILTER_OFF;`
	
	`CAN2SetBaudRate(1,3,3,3,1,can_config_flags);`
	`// set the CAN2 module baud rate`

CAN2SetMask

Prototype	`void CAN2SetMask(unsigned int CAN_MASK, long val, unsigned int CAN_CONFIG_FLAGS);`
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` & `CAN_CONFIG_STD_MSG`, `CAN_CONFIG_MATCH_MSG_TYPE` & `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 & CAN_CONFIG_XTD_MSG);`

CAN2SetFilter

Prototype	`void CAN2SetFilter(unsigned int CAN_FILTER, long val, unsigned int CAN_CONFIG_FLAGS);`
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	`unsigned int CAN2Read(unsigned long *id, char *data, unsigned int *dataLen, unsigned int *CAN_RX_MSG_FLAGS);`
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 `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.`
	`unsigned int msg_rcvd, rx_flags, data_len;`
	`char data[ 8] ;`
	`unsigned long msg_id;`
	`...`
	`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)) {`
	`  ...`
	`}`

CAN2Write

Prototype	`unsigned int CAN2Write(long id, char *Data, unsigned int DataLen, unsigned int CAN_TX_MSG_FLAGS);`
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`
	`unsigned int tx_flags;`
	`char data[ 8] ;`
	`unsigned long msg_id;`
	`...`
	`CAN2SetOperationMode(CAN_MODE_NORMAL,0xFF);`
	`// set NORMAL mode (CAN2 must be in mode in which transmission is`
	`// possible)`
	
	`tx_flags = ECAN_TX_PRIORITY_0 &`
	`ECAN_TX_XTD_FRAME &`
	`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 these as their argument:

```
const unsigned int
 CAN_MODE_BITS         = 0x00E0, // Use this to access opmode bits
 CAN_MODE_NORMAL       = 0x00,
 CAN_MODE_DISABLE      = 0x01,
 CAN_MODE_LOOP         = 0x02,
 CAN_MODE_LISTEN       = 0x03,
 CAN_MODE_CONFIG       = 0x04,
 CAN_MODE_LISTEN_ALL   = 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 unsigned int
     CAN_CONFIG_DEFAULT          = 0xFF,    // 11111111

     CAN_CONFIG_PHSEG2_PRG_BIT   = 0x01,
     CAN_CONFIG_PHSEG2_PRG_ON    = 0xFF,    // XXXXXXX1
     CAN_CONFIG_PHSEG2_PRG_OFF   = 0xFE,    // XXXXXXX0

     CAN_CONFIG_LINE_FILTER_BIT = 0x02,
     CAN_CONFIG_LINE_FILTER_ON  = 0xFF,    // XXXXXX1X
     CAN_CONFIG_LINE_FILTER_OFF = 0xFD,    // XXXXXX0X

     CAN_CONFIG_SAMPLE_BIT       = 0x04,
     CAN_CONFIG_SAMPLE_ONCE      = 0xFF,    // XXXXX1XX
     CAN_CONFIG_SAMPLE_THRICE    = 0xFB,    // XXXXX0XX

     CAN_CONFIG_MSG_TYPE_BIT     = 0x08,
     CAN_CONFIG_STD_MSG          = 0xFF,    // XXXX1XXX
     CAN_CONFIG_XTD_MSG          = 0xF7,    // XXXX0XXX
// continues..
```

```
// ..continued

     CAN_CONFIG_DBL_BUFFER_BIT   = 0x10,
     CAN_CONFIG_DBL_BUFFER_ON    = 0xFF,    // XXX1XXXX
     CAN_CONFIG_DBL_BUFFER_OFF   = 0xEF,    // XXX0XXXX

     CAN_CONFIG_MATCH_TYPE_BIT = 0x20,
     CAN_CONFIG_ALL_VALID_MSG  = 0xDF,      // XX0XXXXX
     CAN_CONFIG_MATCH_MSG_TYPE = 0xFF;      // XX1XXXXX
```

You may use bitwise AND (&) to form config byte out of these values. For example:

```
init = CAN_CONFIG_SAMPLE_THRICE &
       CAN_CONFIG_PHSEG2_PRG_ON &
       CAN_CONFIG_STD_MSG       &
       CAN_CONFIG_DBL_BUFFER_ON &
       CAN_CONFIG_VALID_XTD_MSG &
       CAN_CONFIG_LINE_FILTER_OFF;
...
CANInitialize(1, 1, 3, 3, 1, init);    // initialize CAN
```

CAN_TX_MSG_FLAGS

CAN_TX_MSG_FLAGS are flags related to transmission of a CAN message:

```
const unsigned int
     CAN_TX_PRIORITY_BITS = 0x03,
     CAN_TX_PRIORITY_0    = 0xFC,    // XXXXXX00
     CAN_TX_PRIORITY_1    = 0xFD,    // XXXXXX01
     CAN_TX_PRIORITY_2    = 0xFE,    // XXXXXX10
     CAN_TX_PRIORITY_3    = 0xFF,    // XXXXXX11

     CAN_TX_FRAME_BIT     = 0x08,
     CAN_TX_STD_FRAME     = 0xFF,    // XXXXX1XX
     CAN_TX_XTD_FRAME     = 0xF7,    // XXXXX0XX

     CAN_TX_RTR_BIT       = 0x40,
     CAN_TX_NO_RTR_FRAME  = 0xFF,    // X1XXXXXX
     CAN_TX_RTR_FRAME     = 0xBF;    // X0XXXXXX
```

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

```
/* form value to be used with CANSendMessage: */
send_config = CAN_TX_PRIORITY_0 &
              CAN_TX_XTD_FRAME  &
              CAN_TX_NO_RTR_FRAME;
...
CANSendMessage(id, data, 1, send_config);
```

CAN_RX_MSG_FLAGS

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

```
const unsigned int
     CAN_RX_FILTER_BITS = 0x0007,    // Use this to access filter
                                     // bits
     CAN_RX_FILTER_1     = 0x00,
     CAN_RX_FILTER_2     = 0x01,
     CAN_RX_FILTER_3     = 0x02,
     CAN_RX_FILTER_4     = 0x03,
     CAN_RX_FILTER_5     = 0x04,
     CAN_RX_FILTER_6     = 0x05,

     CAN_RX_OVERFLOW     = 0x08,    // Set if Overflowed else
                                    // cleared
     CAN_RX_INVALID_MSG = 0x10,    // Set if invalid else
                                    // cleared
     CAN_RX_XTD_FRAME    = 0x20,    // Set if XTD message else
                                    // cleared
     CAN_RX_RTR_FRAME     = 0x40,   // Set if RTR message else
                                    // cleared
     CAN_RX_DBL_BUFFERED = 0x80;   // Set if this message was
                                    // hardware double-buffered
```

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

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

CAN_MASK

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

```
const unsigned int
     CAN_MASK_B1 = 0,
     CAN_MASK_B2 = 1;
```

CAN_FILTER

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

```
const unsigned int
     CAN_FILTER_B1_F1 = 0,
     CAN_FILTER_B1_F2 = 1,
     CAN_FILTER_B2_F1 = 2,
     CAN_FILTER_B2_F2 = 3,
     CAN_FILTER_B2_F3 = 4,
     CAN_FILTER_B2_F4 = 5;
```

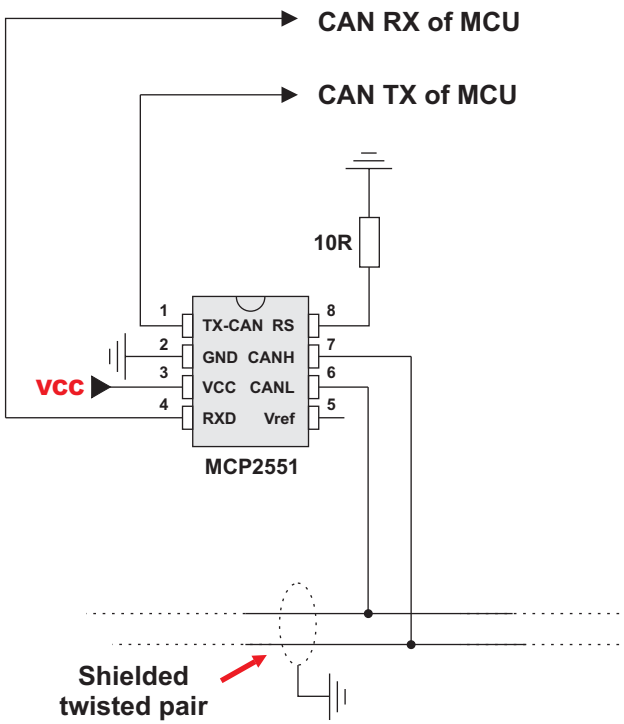
Library Example

The code is a simple demonstration of CAN Library routines usage.

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

void main() {

  ADPCFG = 0xFFFF;
  PORTB = 0;
  TRISB = 0;

  aa = 0;
  aa1 = 0;
  aa2 = 0;

  aa1 =    CAN_TX_PRIORITY_0 &                    // Form value to be used
           CAN_TX_XTD_FRAME &                      //  with CAN1Write
           CAN_TX_NO_RTR_FRAME;

  aa =     CAN_CONFIG_SAMPLE_THRICE &             // Form value to be used
           CAN_CONFIG_PHSEG2_PRG_ON &             //  with CAN1Initialize
           CAN_CONFIG_XTD_MSG &
           CAN_CONFIG_DBL_BUFFER_ON &
           CAN_CONFIG_MATCH_MSG_TYPE &
           CAN_CONFIG_LINE_FILTER_OFF;

  data[ 0]  = 9;
  CAN1Initialize(1,3,3,3,1,aa);                   // initialize CAN
  CAN1SetOperationMode(CAN_MODE_CONFIG,0xFF);     // set CONFIGURATION mode
  id = -1;

//continues...
```

```
//..continued

  CAN1SetMask(CAN_MASK_B1,id,CAN_CONFIG_MATCH_MSG_TYPE & CAN_CONFIG_XTD_MSG);
  // set all mask1 bits to ones
  CAN1SetMask(CAN_MASK_B2,id,CAN_CONFIG_MATCH_MSG_TYPE & CAN_CONFIG_XTD_MSG);
  // set all mask2 bits to ones
  CAN1SetFilter(CAN_FILTER_B2_F3,3,CAN_CONFIG_XTD_MSG);
  // set id of filter B1_F1 to 3
  CAN1SetOperationMode(CAN_MODE_NORMAL,0xFF);       // set NORMAL mode
  id = 12111;
  CAN1Write(id,data,1,aa1);
  while (1) {
    zr = CAN1Read(&id , data , &len, &aa2);
    if ((id == 3u) && zr) {
      PORTB = data[ 0];                             // output data at portB
      data[ 0] ++ ;
      Delay_ms(10);
      id = 12111;
      CAN1Write(id, data, 1,aa1);                  // send incremented data back
    }
  }
} //~!
```

Hardware Connection



CANSPI Library

The SPI module is available with a number of the dsPIC30/33 and PIC24 MCUs. The mikroC 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 the mikroC 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
CANSPI1Init                    CANSPI2Init
CANSPI1SetBaudRate             CANSPI2SetBaudRate
CANSPI1SetMask                 CANSPI2SetMask
CANSPI1SetFilter               CANSPI2SetFilter
CANSPI1Read                    CANSPI2Read
CANSPI1Write                   CANSPI2Write
```

Following routines are for the internal use by compiler only:

```
RegsToCANSPI1ID, CANSPI2IDToRegs
CANSPI1IDToRegs, RegsToCANSPI2ID
```

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

CANSPI1SetOperationMode

Prototype	`void CANSPI1SetOperationMode(char mode, char WAIT);`
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. 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 connection example 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	`char CANSPI1GetOperationMode(void);`
Returns	Current operation mode.
Description	The function returns current operation mode of the CANSPI1 module. Check `CANSPI_OP_MODE` constants (see CANSPI constants) or device datasheet for operation mode codes.

Example	`// check whether the CANSPI1 module is in Normal mode and if it`
	`// is do something.`
	`if (CANSPI1GetOperationMode() == CANSPI_MODE_NORMAL) {`
	`  ...`
	`}`

CANSPI1Init

Prototype	`void CANSPI1Init( char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS, unsigned int * RstPort, char RstPin, unsigned int * CSPort, char CSPin);`
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 given parameters - CAN mode : Normal - Filter and mask registers IDs are set to zero - Filter and mask message frame type is set according to `CAN_CONFIG_FLAGS` value `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to `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 connection example at the bottom of this page.

Example	`// initialize the CANSPI1 module with the appropriate baud rate`
	`// and message acceptance flags along with the sampling rules`
	`unsigned int can_config_flags;`
	`...`
	`can_config_flags = CANSPI_CONFIG_SAMPLE_THRICE &`
	`CANSPI_CONFIG_PHSEG2_PRG_ON &`
	`CANSPI_CONFIG_STD_MSG &`
	`CANSPI_CONFIG_DBL_BUFFER_ON &`
	`CANSPI_CONFIG_VALID_XTD_MSG &`
	`CANSPI_CONFIG_LINE_FILTER_OFF;`
	`...`
	`CANSPI1Initialize(1, 1, 3, 3, 1, can_config_flags);`
	`// initialize CANSPI1`

CANSPI1SetBaudRate

Prototype	`void CANSPI1SetBaudRate( char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);`
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 `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 CAN-SPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// set required baud rate and sampling rules`
	`unsigned int can_config_flags;`
	`...`
	`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 &`
	`CANSPI_CONFIG_PHSEG2_PRG_ON &`
	`CANSPI_CONFIG_STD_MSG &`
	`CANSPI_CONFIG_DBL_BUFFER_ON &`
	`CANSPI_CONFIG_VALID_XTD_MSG &`
	`CANSPI_CONFIG_LINE_FILTER_OFF;`
	
	`CANSPI1SetBaudRate(1, 1, 3, 3, 1, can_config_flags);`

CANSPI1SetMask

Prototype	`void CANSPI1SetMask(unsigned short CAN_MASK, long value, unsigned short CAN_CONFIG_FLAGS);`
Description	Configures mask for advanced filtering of messages. The parameter value is bit-adjusted to the appropriate mask registers. Parameters: - `CAN_MASK`: CANSPI1 module mask number. Valid values: `CANSPI_MASK` constants (see CANSPI constants) - `value`: mask register value - `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values: `CANSPI_CONFIG_ALL_VALID_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE & CANSPI_CONFIG_STD_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE & 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 CAN-SPI Extra Board or similar hardware. See connection example 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 & CANSPI_CONFIG_XTD_MSG);`

CANSPI1SetFilter

Prototype	`void CANSPI1SetFilter(unsigned short CAN_FILTER, long value, unsigned short CAN_CONFIG_FLAGS);`

Description	Configures message filter. The parameter value is bit-adjusted to the appropriate filter registers. Parameters:
	
	- `CAN_FILTER`: CANSPI1 module filter number.
	Valid values: `CANSPI_FILTER` constants (see CANSPI constants)
	- `value`: filter register value
	- `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values:
	`CANSPI_CONFIG_ALL_VALID_MSG,`
	`CANSPI_CONFIG_MATCH_MSG_TYPE & CANSPI_CONFIG_STD_MSG,`
	`CANSPI_CONFIG_MATCH_MSG_TYPE & 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 connection example 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	`unsigned short CANSPI1Read(long *id, unsigned short *data, unsigned short *datalen, unsigned short *CAN_RX_MSG_FLAGS);`
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 connection example at the bottom of this page.

Example	`// check the CANSPI1 module for received messages. If any was`
	`// received do something.`
	`unsigned int msg_rcvd, rx_flags, data_len;`
	`char data[8];`
	`unsigned long msg_id;`
	`...`
	`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)) {`
	`  ...`
	`}`

CANSPI1Write

Prototype	`unsigned short CANSPI1Write(long id, unsigned short *data, unsigned short datalen, unsigned short CAN_TX_MSG_FLAGS);`
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 connection example at the bottom of this page.

Example	`// send message extended CAN message with the appropriate ID and // data`
	`unsigned int tx_flags;`
	`char data[ 8] ;`
	`long msg_id;`
	`...`
	`CANSPI1SetOperationMode(CAN_MODE_NORMAL,0xFF);`
	`// set NORMAL mode (CANSPI1 must be in mode in which transmission // is possible)`
	
	`tx_flags = CANSPI_TX_PRIORITY_0 & CANSPI_TX_XTD_FRAME;`
	`// set message flags`
	`CANSPI1Write(msg_id, data, 2, tx_flags);`

CANSPI2SetOperationMode

Prototype	`void CANSPI2SetOperationMode(char mode, char WAIT);`
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. 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 connection example 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	`char CANSPI2GetOperationMode(void);`
Returns	Current operation mode.
Description	The function returns current operation mode of the CANSPI2 module.
Requires	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 connection example 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) {`
	`  ...`
	`}`

CANSPI2Init

Prototype	`void CANSPI2Init( char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS, unsigned int * RstPort, char RstPin, unsigned int * CSPort, char CSPin);`
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 given parameters - CAN mode : Normal - Filter and mask registers IDs are set to zero - Filter and mask message frame type is set according to CAN_CONFIG_FLAGS value `SAM`, `SEG2PHTS`, `WAKFIL` and `DBEN` bits are set according to `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 connection example at the bottom of this page.

Example	`// initialize the CANSPI2 module with the appropriate baud rate`
	`// and message acceptance flags along with the sampling rules`
	`unsigned int can_config_flags;`
	`...`
	`can_config_flags = CANSPI_CONFIG_SAMPLE_THRICE &`
	`CANSPI_CONFIG_PHSEG2_PRG_ON &`
	`CANSPI_CONFIG_STD_MSG &`
	`CANSPI_CONFIG_DBL_BUFFER_ON &`
	`CANSPI_CONFIG_VALID_XTD_MSG &`
	`CANSPI_CONFIG_LINE_FILTER_OFF;`
	`...`
	`CANSPI2Initialize(1, 1, 3, 3, 1, can_config_flags);`
	`// initialize CANSPI2`

CANSPI2SetBaudRate

Prototype	`void CANSPI2SetBaudRate( char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);`
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 `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 connection example at the bottom of this page.

Example	`// set required baud rate and sampling rules`
	`unsigned int can_config_flags;`
	`...`
	`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 &`
	`CANSPI_CONFIG_PHSEG2_PRG_ON &`
	`CANSPI_CONFIG_STD_MSG &`
	`CANSPI_CONFIG_DBL_BUFFER_ON &`
	`CANSPI_CONFIG_VALID_XTD_MSG &`
	`CANSPI_CONFIG_LINE_FILTER_OFF;`
	`CANSPI2SetBaudRate(1, 1, 3, 3, 1, can_config_flags);`

CANSPI2SetMask

Prototype	`void CANSPI2SetMask(unsigned short CAN_MASK, long value, unsigned short CAN_CONFIG_FLAGS);`

Description	Configures mask for advanced filtering of messages. The parameter value is bit-adjusted to the appropriate mask registers.
	
	Parameters:
	- CAN_MASK: CANSPI2 module mask number.
	Valid values: CANSPI_MASK constants (see CANSPI constants)
	- value: mask register value
	- CAN_CONFIG_FLAGS: selects type of message to filter. Valid values:
	`CANSPI_CONFIG_ALL_VALID_MSG,`
	`CANSPI_CONFIG_MATCH_MSG_TYPE & CANSPI_CONFIG_STD_MSG,`
	`CANSPI_CONFIG_MATCH_MSG_TYPE & 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 connection example 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 & CANSPI_CONFIG_XTD_MSG);`

CANSPI2SetFilter

Prototype	`void CANSPI2SetFilter(unsigned short CAN_FILTER, long value, unsigned short CAN_CONFIG_FLAGS);`

Description	Configures message filter. The parameter value is bit-adjusted to the appropriate filter registers.
	
	Parameters:
	- `CAN_FILTER`: CANSPI2 module filter number.
	Valid values: CANSPI_FILTER constants (see CANSPI constants)
	- `value`: filter register value
	- `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values:
	`CANSPI_CONFIG_ALL_VALID_MSG,`
	`CANSPI_CONFIG_MATCH_MSG_TYPE & CANSPI_CONFIG_STD_MSG,`
	`CANSPI_CONFIG_MATCH_MSG_TYPE & 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 connection example 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	`unsigned short CANSPI2Read(long *id, unsigned short *data, unsigned short *datalen, unsigned short *CAN_RX_MSG_FLAGS);`
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 connection example at the bottom of this page.

Example	`// check the CANSPI2 module for received messages. If any was`
	`// received do something.`
	`unsigned int msg_rcvd, rx_flags, data_len;`
	`char data[ 8] ;`
	`unsigned long msg_id;`
	`...`
	`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)) {`
	`  ...`
	`}`

CANSPI2Write

Prototype	`unsigned short CANSPI2Write(long id, unsigned short *data, unsigned short datalen, unsigned short CAN_TX_MSG_FLAGS);`
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 connection example at the bottom of this page.

Example	`// send message extended CAN message with the appropriate ID and // data`
	`unsigned int tx_flags;`
	`char data[ 8] ;`
	`long msg_id;`
	`...`
	`CANSPI2SetOperationMode(CAN_MODE_NORMAL,0xFF);`
	`// set NORMAL mode (CANSPI2 must be in mode in which transmission // is possible)`
	
	`tx_flags = CANSPI_TX_PRIORITY_0 & 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 these as their argument:

```
const char
     CANSPI_MODE_BITS    = 0xE0, // Use this to access opmode  bits
     CANSPI_MODE_NORMAL  = 0x00,
     CANSPI_MODE_SLEEP   = 0x20,
     CANSPI_MODE_LOOP    = 0x40,
     CANSPI_MODE_LISTEN  = 0x60,
     CANSPI_MODE_CONFIG  = 0x80;
```

CANSPI_CONFIG_FLAGS

The CANSPI_CONFIG_FLAGS constants define flags related to the CANSPI module configuration. The functions CANSPI1Init, CANSPI2Init, CANSPI1SetBaudRate, CANSPI2SetBaudRate, CANSPI1SetMask, CANSPI2SetMask, CANSPI1SetFilter and CANSPI2SetFilter expect one of these (or a bitwise combination) as their argument:

```
const char
     CANSPI_CONFIG_DEFAULT          = 0xFF,    // 11111111
     CANSPI_CONFIG_PHSEG2_PRG_BIT   = 0x01,
     CANSPI_CONFIG_PHSEG2_PRG_ON    = 0xFF,    // XXXXXXX1
     CANSPI_CONFIG_PHSEG2_PRG_OFF   = 0xFE,    // XXXXXXX0

     CANSPI_CONFIG_LINE_FILTER_BIT = 0x02,
     CANSPI_CONFIG_LINE_FILTER_ON   = 0xFF,    // XXXXXX1X
     CANSPI_CONFIG_LINE_FILTER_OFF = 0xFD,    // XXXXXX0X

     CANSPI_CONFIG_SAMPLE_BIT       = 0x04,
     CANSPI_CONFIG_SAMPLE_ONCE      = 0xFF,    // XXXXX1XX
     CANSPI_CONFIG_SAMPLE_THRICE    = 0xFB,    // XXXXX0XX

     CANSPI_CONFIG_MSG_TYPE_BIT     = 0x08,
     CANSPI_CONFIG_STD_MSG          = 0xFF,    // XXXX1XXX
     CANSPI_CONFIG_XTD_MSG          = 0xF7,    // XXXX0XXX

// continues..
```

```
// ..continued
     CANSPI_CONFIG_DBL_BUFFER_BIT   = 0x10,
     CANSPI_CONFIG_DBL_BUFFER_ON    = 0xFF,    // XXX1XXXX
     CANSPI_CONFIG_DBL_BUFFER_OFF   = 0xEF,    // XXX0XXXX

     CANSPI_CONFIG_MSG_BITS          = 0x60,
     CANSPI_CONFIG_ALL_MSG           = 0xFF,    // X11XXXXX
     CANSPI_CONFIG_VALID_XTD_MSG    = 0xDF,    // X10XXXXX
     CANSPI_CONFIG_VALID_STD_MSG    = 0xBF,    // X01XXXXX
     CANSPI_CONFIG_ALL_VALID_MSG    = 0x9F;    // X00XXXXX
```

You may use bitwise AND (&) to form config byte out of these values. For example:

```
init = CANSPI_CONFIG_SAMPLE_THRICE &
        CANSPI_CONFIG_PHSEG2_PRG_ON &
        CANSPI_CONFIG_STD_MSG        &
        CANSPI_CONFIG_DBL_BUFFER_ON &
        CANSPI_CONFIG_VALID_XTD_MSG &
        CANSPI_CONFIG_LINE_FILTER_OFF;
...
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 char
     CANSPI_TX_PRIORITY_BITS = 0x03,
     CANSPI_TX_PRIORITY_0    = 0xFC,    // XXXXXX00
     CANSPI_TX_PRIORITY_1    = 0xFD,    // XXXXXX01
     CANSPI_TX_PRIORITY_2    = 0xFE,    // XXXXXX10
     CANSPI_TX_PRIORITY_3    = 0xFF,    // XXXXXX11

     CANSPI_TX_FRAME_BIT     = 0x08,
     CANSPI_TX_STD_FRAME     = 0xFF,     // XXXXX1XX
     CANSPI_TX_XTD_FRAME     = 0xF7,     // XXXXX0XX

     CANSPI_TX_RTR_BIT       = 0x40,
     CANSPI_TX_NO_RTR_FRAME = 0xFF,     // X1XXXXXX
     CANSPI_TX_RTR_FRAME     = 0xBF;     // X0XXXXXX
```

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

```
/* form value to be used as sending message flag : */
send_config = CANSPI_TX_PRIORITY_0 &
               CANSPI_TX_XTD_FRAME  &
               CANSPI_TX_NO_RTR_FRAME;
...
CANSPI1Write(id, data, 1, send_config);
```

CANSPI_RX_MSG_FLAGS

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

```
const char
     CANSPI_RX_FILTER_BITS = 0x07,//Use this to access filter bits
     CANSPI_RX_FILTER_1    = 0x00,
     CANSPI_RX_FILTER_2    = 0x01,
     CANSPI_RX_FILTER_3    = 0x02,
     CANSPI_RX_FILTER_4    = 0x03,
     CANSPI_RX_FILTER_5    = 0x04,
     CANSPI_RX_FILTER_6    = 0x05,

     CANSPI_RX_OVERFLOW    = 0x08,//Set if Overflowed else cleared
     CANSPI_RX_INVALID_MSG = 0x10,//Set if invalid else cleared
     CANSPI_RX_XTD_FRAME   = 0x20,//Set if XTD message else cleared
     CANSPI_RX_RTR_FRAME   = 0x40,//Set if RTR message else cleared
     CANSPI_RX_DBL_BUFFERED = 0x80;
     //Set if this message was hardware double-buffered
```

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

```
if (MsgFlag & CANSPI_RX_OVERFLOW != 0) {
   ...
   // Receiver overflow has occurred.
   // We have lost our previous message.
}
```

CANSPI_MASK

The CANSPI_MASK constants define mask codes. The functions CANSPI1SetMask and CANSPI1SetMask expect one of these as their argument:

```
const char
     CANSPI_MASK_B1 = 0,
     CANSPI_MASK_B2 = 1;
```

CANSPI_FILTER

The CANSPI_FILTER constants define filter codes. The functions CANSPI1SetFilter and CANSPI2SetFilter expects one of these as their argument:

```
const char
     CANSPI_FILTER_B1_F1 = 0,
     CANSPI_FILTER_B1_F2 = 1,
     CANSPI_FILTER_B2_F1 = 2,
     CANSPI_FILTER_B2_F2 = 3,
     CANSPI_FILTER_B2_F3 = 4,
     CANSPI_FILTER_B2_F4 = 5;
```

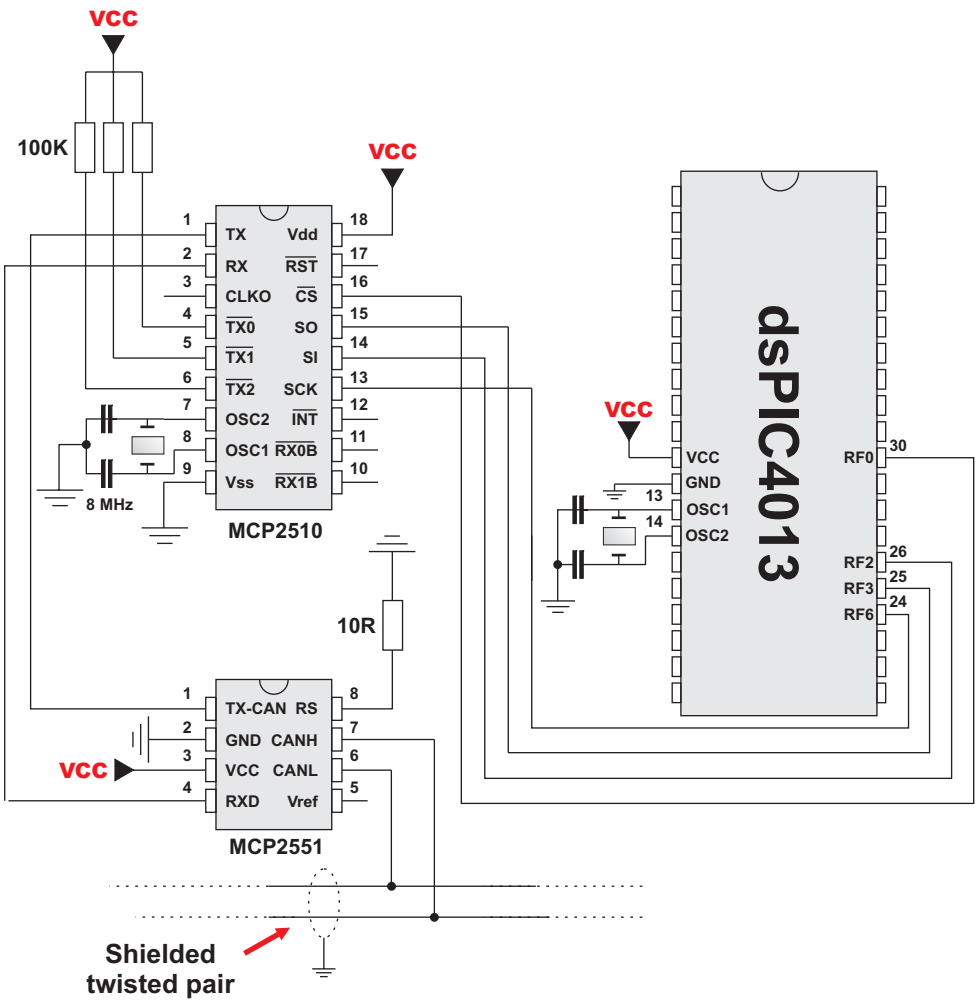
Library Example

The code is a simple demonstration of CANSPI Library routines usage.

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

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

  aa = 0;
  aa1 = 0;
  aa2 = 0;
  aa =  CANSPI_CONFIG_SAMPLE_THRICE &   // form value to be used
        CANSPI_CONFIG_PHSEG2_PRG_ON &   //      with CANSPI1Init
        CANSPI_CONFIG_XTD_MSG &
        CANSPI_CONFIG_DBL_BUFFER_ON &
        CANSPI_CONFIG_VALID_XTD_MSG;
  aa1 = CANSPI_TX_PRIORITY_0 &          // form value to be used
        CANSPI_TX_XTD_FRAME &           //      with CANSPI1Write
        CANSPI_TX_NO_RTR_FRAME;

  Spi1_Init();
  CANSPI1Init(1,3,3,3,1,aa, &PORTF, 1, &PORTF, 0);
  // initialize the external CAN module

// continues ..
```

```
// .. continued

  CANSPI1SetOperationMode(CANSPI_MODE_CONFIG,0xFF); // set CONFIGURATION mode

  ID=-1;
  CANSPI1SetMask(CANSPI_MASK_B1,id,CANSPI_CONFIG_XTD_MSG);
  // set all mask1 bits to ones
  CANSPI1SetMask(CANSPI_MASK_B2,id,CANSPI_CONFIG_XTD_MSG);
  // set all mask2 bits to ones
  CANSPI1SetFilter(CANSPI_FILTER_B2_F4,3,CANSPI_CONFIG_XTD_MSG);
  // set id of filter B1_F1 to 3
  CANSPI1SetOperationMode(CANSPI_MODE_NORMAL,0xFF);
  // set NORMAL mode

  data[ 0]  = 7;
  id = 12111;
  CANSPI1Write(id,data,1,aa1);
  while(1) {
    zr = CANSPI1Read(&id , data , &len, &aa2); // receive data, if any
    if ((id == 3) && (zr)) {
      PORTB = data[ 0] ;
      data[ 0] ++;                          // output data at portB
      id=12111;
      delay_ms(100);
      CANSPI1Write(id,data,1,aa1);
      if (len == 2) {                       // send incremented data back
        PORTD = data[ 1] ;                  // if message contains two data bytes
      }                                     // output second byte at portd
    }
  }
} //~!
```

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 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.

Notes: - Routines for file handling can be used only with FAT16 file system.
- 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 CF card has Master Boot Record (MBR), the library will work with the first available primary (logical) partition that has non-zero size. If CF 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_Read_Word
Cf_Write_Init
Cf_Write_Byte
Cf_Write_Word
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	`void Cf_Init(unsigned *port_A012, unsigned pin_A0, unsigned pin_A1, unsigned pin_A2, unsigned *port_RDY, unsigned pin_RDY, unsigned *port_WE, unsigned pin_WE, unsigned *port_OE, unsigned pin_OE, unsigned *port_CD1, unsigned pin_CD1, unsigned *port_CE1, unsigned pin_CE1, unsigned *port_Data);`
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`
	`void Cf_Init_EASYdsPIC2() {`
	`  Cf_Init(&PORTB,8,9,10, &PORTC,15, &PORTC,14, &PORTC,13, &PORTB,12, &PORTB,11, &PORTB);`
	`} //~`
	
	`//-------------- Init for dsPICPRO`
	`void Cf_Init_dsPICPRO2() {`
	`  Cf_Init(&PORTD,8,9,10, &PORTG,14, &PORTG,12, &PORTD,11, &PORTG,15, &PORTG,13, &PORTD);`
	`} //~`

Cf_Detect

Prototype	`unsigned Cf_Detect(void);`
Returns	`1` - if CF card was detected `0` - otherwise
Description	Checks for presence of CF card by reading the chip detect pin. **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:`
	`do`
	`  asm nop;`
	`while (!Cf_Detect());`

Cf_Enable

Prototype	`void Cf_Enable(void);`
Description	Enables the device. Routine needs to be called only if you have disabled the device by means of the `Cf_Disable` routine. These two routines in conjunction allow you to free/occupy data line when working with multiple devices.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`.
Example	`// enable compact flash` `Cf_Enable();`

Cf_Disable

Prototype	`void Cf_Disable(void);`
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	`Cf_Disable();`

Cf_Read_Init

Prototype	`void Cf_Read_Init(unsigned long address, unsigned short sectcnt);`
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	`unsigned Cf_Read_Byte(void);`
Returns	Returns a byte read from CF. **Note**: Higher byte of the unsigned return value is cleared.
Description	Reads one byte from CF location currently pointed to by internal read pointers. These pointers will be autoicremented after complete reading.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`. CF card must be initialized for reading operation. See `Cf_Read_Init`.

Example	`// Read a byte from compact flash:`
	`char data;`
	`...`
	`data = Cf_Read_Byte();`

Cf_Read_Word

Prototype	`unsigned Cf_Read_Word(void);`
Returns	Returns a word (16-bit) read from CF.
Description	Reads one word from CF location currently pointed to by internal read pointers. These pointers will be autoicremented by 2 after complete reading. The reading process is implemented as two sequential byte reads. A byte read from lower address is placed into lower byte of the result. A byte read from higher address is placed into higher byte of the result.
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 word and display it on PORTC:` `PORTC = Cf_Read_Word();`

Cf_Write_Init

Prototype	`void Cf_Write_Init(unsigned long address, unsigned short sectcnt);`
Description	Initializes CF card for writing. Parameters : - `address`: the first sector to be prepared for writing operation. - `sectcnt`: number of sectors to be prepared for writing operation.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`.
Example	`// initialize compact flash for writing to sector 590` `Cf_Write_Init(590, 1);`

Cf_Write_Byte

Prototype	`void Cf_Write_Byte(unsigned short data);`
Description	Writes a byte to CF location currently pointed to by writing pointers. These pointers will be autoicremented after complete reading. Parameters : - `data`: byte to be written.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`. CF card must be initialized for writing operation. See `Cf_Write_Init`.

Example	`char data = 0xAA;`
	`...`
	`Cf_Write_Byte(data);`

Cf_Write_Word

Prototype	`void Cf_Write_Word(unsigned data);`
Returns	Nothing.
Description	Writes a word to CF location currently pointed to by internal writing pointers. These pointers will be autoicremented by 2 after complete reading. Writing process is implemented as two sequential byte writes. Lower byte of data is placed into CF location with lower address. Higher byte of data is placed into CF location with higher address. Parameters : - `data`: word 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	`unsigned data = 0xAAAA;`
	`...`
	`Cf_Write_Word(data);`

Cf_Read_Sector

Prototype	`void Cf_Read_Sector(unsigned sector_number, unsigned char *buffer);`
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`
	`char data[ 512];`
	`...`
	`Cf_Read_Sector(22, data);`

Cf_Write_Sector

Prototype	`void Cf_Write_Sector(unsigned sector_number, unsigned char *buffer);`
Returns	Nothing.
Description	Writes 512 bytes of data provided by the buffer parameter to one CF sector. Parameters : `sector_number`: sector to be written to. `buffer`: data buffer of 512 bytes in length.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See `Cf_Init`.

Example	`// write to sector 22`
	`char data[ 512];`
	`...`
	`Cf_Write_Sector(22, data);`

Cf_Fat_Init

Prototype	`unsigned short Cf_Fat_Init(unsigned *port_A012, unsigned pin_A0, unsigned pin_A1, unsigned pin_A2, unsigned *port_RDY, unsigned pin_RDY, unsigned *port_WE, unsigned pin_WE, unsigned *port_OE, unsigned pin_OE, unsigned *port_CD1, unsigned pin_CD1, unsigned *port_CE1, unsigned pin_CE1, unsigned *port_Data);`
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 necessary 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))`
	`{`
	`  ...`
	`}`

Cf_Fat_QuickFormat

Prototype	`unsigned short Cf_Fat_QuickFormat(unsigned *port_A012, unsigned pin_A0, unsigned pin_A1, unsigned pin_A2, unsigned *port_RDY, unsigned pin_RDY, unsigned *port_WE, unsigned pin_WE, unsigned *port_OE, unsigned pin_OE, unsigned *port_CD1, unsigned pin_CD1, unsigned *port_CE1, unsigned pin_CE1, unsigned *port_Data, char *cf_fat_label);`
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. **Note**: This routine can be used instead or in conjunction with `Cf_Fat_Init` routine. **Note**: If CF card already contains a valid boot sector, it will remain unchanged (except volume label field) and only FAT and ROOT tables will be erased. Also, the new volume label will be set.
Requires	Nothing

Example	`//--- format and initialize the FAT library - dsPICPRO2`
	`if (!Cf_Fat_Init(&PORTD,8,9,10, &PORTG,14, &PORTG,12, &PORTD,11, &PORTG,15, &PORTG,13, &PORTD, "mikroE"))`
	`{`
	`  ...`
	`}`

Cf_Fat_Assign

Prototype	`unsigned Cf_Fat_Assign(char *filename, char file_cre_attr);`
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 over the assigned file. Parameters : - `filename`: name of the file that should be assigned for file operations. The file name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between the file name and extension (i.e. "MIKROELETXT" -> MIKROELE.TXT). In this case the last 3 characters of the string are considered to be file extension. - `file_cre_attr`: file creation and attributs flags. Each bit corresponds to the appropriate file attribut:

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	`void Cf_fat_Reset(unsigned long *size);`
Returns	Nothing.
Description	Opens currently assigned file for reading. Parameters : - `size`: buffer to store file size to. After file has been open for reading its size is returned through this parameter.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. File must be previously assigned. See `Cf_Fat_Assign`.

Example	`unsigned long size;`
	`...`
	`Cf_Fat_Reset(size);`

Cf_Fat_Read

Prototype	`void Cf_Fat_Read(unsigned char *bdata);`
Description	Reads a byte from currently assigned file opened for reading. Upon function execution file pointers will be set to the next character in the file. Parameters : `bdata`: buffer to store read byte to. Upon this function execution read byte is returned through this parameter.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. File must be previously assigned. See `Cf_Fat_Assign`. File must be open for reading. See `Cf_Fat_Reset`.

Example	`char character;`
	`...`
	`Cf_Fat_Read(&character);`

Cf_Fat_Rewrite

Prototype	`void Cf_fat_Rewrite();`
Returns	Nothing.
Description	Opens currently assigned file for writing. If the file is not empty its content will be erased.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. The file must be previously assigned. See `Cf_Fat_Assign`.
Example	`// open file for writing` `Cf_Fat_Rewrite();`

Cf_Fat_Append

Prototype	`void Cf_fat_Append();`
Returns	Nothing.
Description	Opens currently assigned file for appending. Upon this function execution file pointers will be positioned after the last byte in the file, so any subsequent file writing operation will start from there.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. File must be previously assigned. See `Cf_Fat_Assign`.
Example	`// open file for appending` `Cf_Fat_Append();`

Cf_Fat_Delete

Prototype	`void Cf_Fat_Delete();`
Description	Deletes currently assigned file from CF card.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. File must be previously assigned. See `Cf_Fat_Assign`.
Example	`// delete current file` `Cf_Fat_Delete();`

Cf_Fat_Write

Prototype	`void Cf_fat_Write(char *fdata, unsigned data_len);`
Returns	Nothing.
Description	Writes requested number of bytes to currently assigned file opened for writing. Parameters : - `fdata`: data to be written. - `data_len`: number of bytes to be written.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. File must be previously assigned. See `Cf_Fat_Assign`. File must be open for writing. See `Cf_Fat_Rewrite` or `Cf_Fat_Append`.

Example	`char file_contents[ 42] ;`
	`...`
	`Cf_Fat_Write(file_contents, 42);//write data to the assigned file`

Cf_Fat_Set_File_Date

Prototype	`void Cf_fat_Set_File_Date(unsigned int year, unsigned short month, unsigned short day, unsigned short hours, unsigned short mins, unsigned short seconds);`
Returns	Nothing.
Description	Sets the date/time stamp. Any subsequent file writing operation will write this stamp to currently assigned file's time/date attributs. Parameters : - `year`: year attribute. Valid values: 1980-2107 - `month`: month attribute. Valid values: 1-12 - `day`: day attribute. Valid values: 1-31 - `hours`: hours attribute. Valid values: 0-23 - `mins`: minutes attribute. Valid values: 0-59 - `seconds`: seconds attribute. Valid values: 0-59
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. File must be previously assigned. See `Cf_Fat_Assign`. File must be open for writing. See `Cf_Fat_Rewrite` or `Cf_Fat_Append`.
Example	`Cf_Fat_Set_File_Date(2005,9,30,17,41,0);`

Cf_Fat_Get_File_Date

Prototype	`void Cf_fat_Get_File_Date(unsigned int *year, unsigned short *month, unsigned short *day, unsigned short *hours, unsigned short *mins, unsigned short *seconds);`
Description	Reads time/date attributes of currently assigned file. Parameters : - `year`: buffer to store year attribute to. Upon function execution year attribute is returned through this parameter. - `month`: buffer to store month attribute to. Upon function execution month attribute is returned through this parameter. - `day`: buffer to store day attribute to. Upon function execution day attribute is returned through this parameter. - `hours`: buffer to store hours attribute to. Upon function execution hours attribute is returned through this parameter. - `mins`: buffer to store minutes attribute to. Upon function execution minutes attribute is returned through this parameter.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. File must be previously assigned. See `Cf_Fat_Assign`.

Example	`unsigned year;`
	`char month, day, hours, mins;`
	`...`
	`Cf_Fat_Get_File_Date(&year, &month, &day, &hours, &mins);`

Cf_Fat_Get_File_Size

Prototype	`unsigned long Cf_fat_Get_File_Size();`
Returns	Size of the currently assigned file in bytes.
Description	This function reads size of currently assigned file in bytes.
Requires	CF card and CF library must be initialized for file operations. See `Cf_Fat_Init`. File must be previously assigned. See `Cf_Fat_Assign`.

Example	`unsigned long my_file_size;`
	`...`
	`my_file_size = Cf_Fat_Get_File_Size();`

Cf_Fat_Get_Swap_File

Prototype	`unsigned long Cf_Fat_Get_Swap_File(unsigned long sectors_cnt, char *filename, char file_attr);`
Returns	Number of the start sector for the newly created swap file, if there was enough free space on 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. Swap file can be considered as a "window" on the media where the user can freely write/read data. It's main purpose in the mikroC's library is to be used for fast data acquisition; when the time-critical acquisition has finished, the data can be re-written into a "normal" file, and formatted in the most suitable way. Parameters: - `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 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

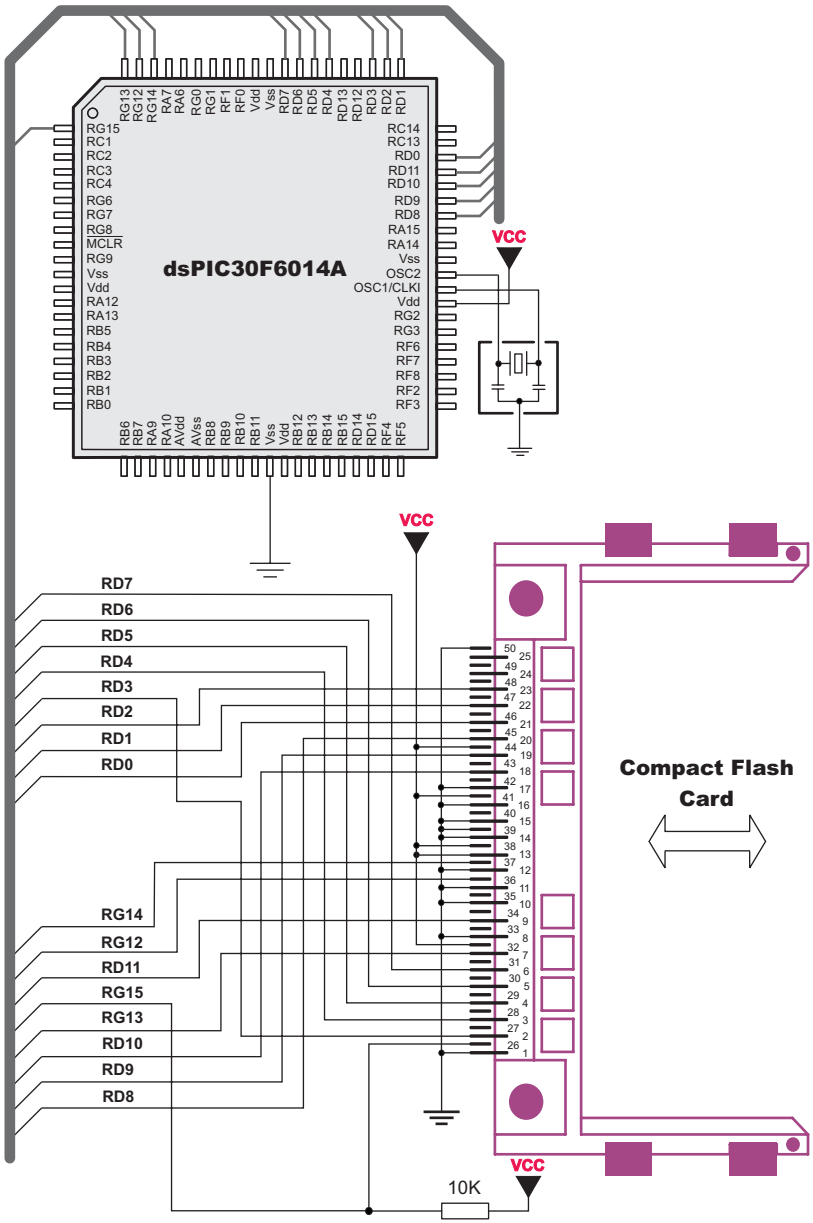
```
// Try to create a swap file with archive atribute, whose size
// will be at least 1000 sectors.
// If it succeeds, it sends the No. of start sector over USART
unsigned long size;
...
size = Cf_Fat_Get_Swap_File(1000, "mikroE.txt", 0x20);
if (size) {
  Usart_Write(0xAA);
  Usart_Write(Lo(size));
  Usart_Write(Hi(size));
  Usart_Write(Higher(size));
  Usart_Write(Highest(size));
  Usart_Write(0xAA);
} //~
```

Library Example

The following example writes 512 bytes at sector no.620, and then reads the data and sends it over UART1 for a visual check. Hardware configurations in this example are made for the dsPICPRO2 board and dsPIC30F6014A.

```
char buff[ 512] ;
//-------------- Init for dsPICPRO2
void Cf_Init_dsPICPRO2() {
  Cf_Init(&PORTD,8,9,10, &PORTG,14, &PORTG,12, &PORTD,11, &PORTG,15, &PORTG,13,
&PORTD);

} //~
void initCF() {
  ADPCFG = 0xFFFF;
  Cf_Init_dsPICPRO2();
  //--- CD1 does not work on non-TTL inputs
  //while (CF_Detect() == 0) ;       // wait until CF card is inserted
  //Delay_ms(500);          // wait for a while until the card is stabilized
} //~
//--------------
void testBytes() {
  unsigned int i, tmp;
  //--- write some data
  CF_Write_Init(620,1);              // Initialize writing at sector address 620
                                     // for 1 sector (byte)
  Uart1_Write_Char('s');             // Notify that writing has started
  Delay_ms(1000);
  for (i=0; i<=511; i++) {           // Write 512 bytes to sector 590
    CF_Write_Byte(i);
  }
  Delay_ms(1000);
  //--- read written data
  CF_Read_Init(620,1);               // Initialize read from sector address 620
  Delay_ms(1000);
                                     //    for 1 sector (byte)
  Cf_Read_Sector(620, buff);
  for (i=0; i<=511; i++) {           // Read 512 bytes from initialized sector
    Uart1_Write_Char(buff[ i] );
 }
} //~
//-------------- Main program
void main() {
  Uart1_Init(19200);

  initCF();
  testBytes();
} //~!
```

Hardware Connection



DSP (Digital Signal Processing) Library

mikroC for dsPIC30/33 and PIC24 includes a libraries for DSP engine.

Library Routines

```
FIR_Radix
IIR_Radix
FFT
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	`unsigned FIR_Radix(unsigned FilterOrder, const unsigned *ptrCoeffs, unsigned BuffLength, unsigned *ptrInput, unsigned Index);`
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	`unsigned IIR_Radix (const int BScale, const int AScale, const signed *ptrB, const signed *ptrA, unsigned FilterOrder, unsigned *ptrInput, unsigned Input_Len, unsigned *ptrOutput, unsigned Index);`
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	`void Fft(unsigned log2N, const unsigned *TwiddleFactorsAddress, unsigned *Samples);`
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 as 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.

BitReverseComplex

Prototype	`void BitReverseComplex(unsigned log2N, unsigned *ReIm);`
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	`void Vector_Set(unsigned *input, unsigned size, unsigned value);`
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	`unsigned VectorPower(unsigned N, unsigned *Vector);`
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	Input samples must be in Y data space.
Note	AccuA used, not restored CORCON saved, used, restored

Vector_Subtract

Prototype	`void Vector_Subtract(unsigned *dest, unsigned *v1, unsigned *v2, unsigned numElems);`
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	`void VectorScale(unsigned N, int ScaleValue, unsigned *SrcVector, unsigned *DestVector);`
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	`void Vector_Negate(unsigned *srcVector, unsigned *DestVector, unsigned numElems);`
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	`void Vector_Multiply(unsigned *v1, unsigned *v2, unsigned *dest, unsigned numElems);`
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	`unsigned Vector_Min(unsigned *Vector, unsigned numElems, unsigned *MinIndex);`
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	`unsigned Vector_Max(unsigned *Vector, unsigned numElems, unsigned *MaxIndex);`
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	`unsigned Vector_Dot(unsigned *v1, unsigned *v2, unsigned numElems);`
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	`void Vector_Correlate(unsigned *v1, unsigned *v2, unsigned *dest, unsigned numElemsV1, unsigned numElemsV2);`
Description	Procedure calculates Vector correlation (using convolution). *v1* - First vector. *v2* - Second vector *numElemsV1* - Number of the first vector elements *numElemsV2* - Number of the 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	`void Vector_Convolve(unsigned *v1, unsigned *v2, unsigned *dest, unsigned numElemsV1, unsigned numElemsV2);`
Description	Procedure calculates Vector using convolution. *v1* - First vector. *v2* - Second vector *numElemsV1* - Number of the first vector elements *numElemsV2* - Number of the 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	`void Vector_Add(unsigned *dest, unsigned *v1, unsigned *v2, unsigned numElems);`
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	`void Matrix_Transpose(unsigned * src, unsigned * dest, unsigned num_rows, unsigned num_cols);`
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	`void Matrix_Subtract(unsigned * src1, unsigned * src2, unsigned * dest, unsigned num_rows, unsigned num_cols);`
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	`void Matrix_Scale(unsigned scale_value, unsigned *src1, unsigned *dest, unsigned num_rows, unsigned num_cols);`
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] = srcM[j][i]**
Note	AccuA used, not restored. CORCON saved, used, restored.

Matrix_Multiply

Prototype	`void Matrix_Multiply(unsigned * src1, unsigned * src2, unsigned * dest, unsigned numRows1, unsigned numCols2, unsigned numCols1Rows2);`
Description	Procedure does matrix multiply. *src1* - First Matrix *src2* - Second Matrix *dest* - Result Matrix *numRows1* - Number of the first matrix rows *numCols2* - Number of the second matrix columns *numCols1Rows2* - Number of the 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	`void Matrix_Add(unsigned * src1, unsigned * src2, unsigned * dest, unsigned numRows, unsigned numCols);`
Description	Procedure does matrix addition. *src1* - First Matrix *src2* - Second Matrix *dest* - Result Matrix *numRows1* - Number of the first matrix rows *numCols2* - Number of the second matrix columns
Operation	**dstM[i][j] = srcM1[i][j] + srcM2[i][j]**
Note	AccuA used, not restored. CORCON saved, used, restored.

ECAN Library (Enhanced Controller Area Network)

The mikroC 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 grater 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.h` 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 via include pragma directive 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.

Library Routines

```
ECAN1DmaChannelInit          ECAN2DmaChannelInit
ECAN1SetOperationMode        ECAN2SetOperationMode
ECAN1GetOperationMode        ECAN2GetOperationMode
ECAN1Initialize              ECAN2Initialize
ECAN1SelectTxBuffers         ECAN2SelectTxBuffers
ECAN1FilterDisable           ECAN2FilterDisable
ECAN1FilterEnable            ECAN2FilterEnable
ECAN1SetBufferSize           ECAN2SetBufferSize
ECAN1SetBaudRate             ECAN2SetBaudRate
ECAN1SetMask                 ECAN2SetMask
ECAN1SetFilter               ECAN2SetFilter
ECAN1Read                    ECAN2Read
ECAN1Write                   ECAN2Write
```

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

```
RegsToECAN1ID, RegsToECAN2ID
ECAN1IDToRegs, ECAN2IDToRegs
```

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

ECAN1DmaChannelInit

Prototype	`unsigned ECAN1DmaChannelInit(unsigned DmaChannel, unsigned ChannelDir, void *DmaRamBuffAdd);`
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. Microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.
Example	`//channel0 will transfer 8 words from dma ram at 0x4000 to ECAN1` `ECAN1DmaChannelInit(0, 1, 0x4000);`

ECAN1SetOperationMode

Prototype	`void ECAN1SetOperationMode(unsigned int mode, unsigned int WAIT);`
Returns	Nothing.
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. 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. 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	`unsigned int ECAN1GetOperationMode(void);`
Returns	Current operation mode.
Description	The function returns current operation mode of the ECAN1 module. Check `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. 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)`
	`{`
	`  ...`
	`}`

ECAN1Initialize

Prototype	`void ECAN1Initialize(unsigned int SJW, unsigned int BRP, unsigned int PHSEG1, unsigned int PHSEG2, unsigned int PROPSEG, unsigned int ECAN_CONFIG_FLAGS);`
Returns	Nothing.
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 `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. Microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.

Example	`// initialize the ECAN1 module with appropriate baud rate and`
	`// message acceptance flags along with the sampling rules`
	`unsigned int ecan_config_flags;`
	`...`
	`ecan_config_flags = ECAN_CONFIG_SAMPLE_THRICE &`
	`// Form value to be used`
	`ECAN_CONFIG_PHSEG2_PRG_ON &`
	`//  with ECANInitialize`
	`ECAN_CONFIG_XTD_MSG &`
	`ECAN_CONFIG_MATCH_MSG_TYPE &`
	`ECAN_CONFIG_LINE_FILTER_OFF;`
	
	`ECAN1Initialize(1, 3, 3, 3, 1, ecan_config_flags);`
	`// initialize the ECAN1 module`

ECAN1SelectTxBuffers

Prototype	`unsigned ECAN1SelectTxBuffers(unsigned txselect);`
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. 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	`void ECAN1FilterDisable(unsigned fltdis);`
Returns	Nothing.
Description	The function disables receive filters. Parameters: - `fltdis`: filter disable selection parameter. Each bit corresponds to appropriate filter. By settung bit the corresponding filter is to be disabled.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. 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	`void ECAN1FilterEnable(unsigned flten);`
Returns	Nothing.
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 ECAN1 routines are supported only by MCUs with the ECAN1 module. 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	`unsigned ECAN1SetBufferSize(unsigned Ecan1BuffSize);`
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.h` header file located in the ECAN project folder.
Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. 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	`void ECAN1SetBaudRate(unsigned int SJW, unsigned int BRP, unsigned int PHSEG1, unsigned int PHSEG2, unsigned int PROPSEG, unsigned int ECAN_CONFIG_FLAGS);`
Returns	Nothing.
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. 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. 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`
	`unsigned int ecan_config_flags;`
	`...`
	`ECAN1SetOperationMode(ECAN_MODE_CONFIG,0xFF);`
	`// set CONFIGURATION mode (ECAN1 module mast be in config mode`
	`// for baud rate settings)`
	
	`ecan_config_flags = ECAN_CONFIG_SAMPLE_THRICE &`
	`// Form value to be used`
	`ECAN_CONFIG_PHSEG2_PRG_ON &`
	`// with ECAN1SetBaudRate`
	`ECAN_CONFIG_XTD_MSG &`
	`ECAN_CONFIG_MATCH_MSG_TYPE &`
	`ECAN_CONFIG_LINE_FILTER_OFF;`
	
	`ECAN1SetBaudRate(1, 3, 3, 3, 1, ecan_config_flags);`
	`// set ECAN1 module baud rate`

ECAN1SetMask

Prototype	`void ECAN1SetMask(unsigned int ECAN_MASK, long val, unsigned int ECAN_CONFIG_FLAGS);`
Returns	Nothing.

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 & ECAN_CONFIG_STD_MSG,`
	`ECAN_CONFIG_MATCH_MSG_TYPE & ECAN_CONFIG_XTD_MSG.`
	(see ECAN constants).

Requires	The ECAN1 routines are supported only by MCUs with the ECAN1 module. 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 & ECAN_CONFIG_XTD_MSG);`

ECAN1SetFilter

Prototype	`void ECAN1SetFilter(unsigned int ECAN_FILTER, long val, unsigned int ECAN_FILTER_MASK, unsigned int ECAN_FILTER_RXBUFF, unsigned int ECAN_CONFIG_FLAGS);`
Returns	Nothing.
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. 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	`unsigned int ECAN1Read(unsigned long *id, char *data, unsigned int *dataLen, unsigned int *ECAN_RX_MSG_FLAGS);`
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. 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.`
	`unsigned int msg_rcvd, rx_flags, data_len;`
	`char data[ 8] ;`
	`unsigned long msg_id;`
	`...`
	`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)) {`
	`  ...`
	`}`

ECAN1Write

Prototype	`unsigned int ECAN1Write(long id, char *Data, unsigned int DataLen, unsigned int ECAN_TX_MSG_FLAGS);`
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. 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`
	`unsigned int tx_flags;`
	`char data[ 8] ;`
	`unsigned long msg_id;`
	`...`
	`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 &`
	`ECAN_TX_XTD_FRAME &`
	`ECAN_TX_NO_RTR_FRAME;`
	`// set message flags`
	`ECAN1Write(msg_id, data, 1, tx_flags);`

ECAN2DmaChannelInit

Prototype	`unsigned ECAN2DmaChannelInit(unsigned DmaChannel, unsigned ChannelDir, void *DmaRamBuffAdd);`
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. 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	`void ECAN2SetOperationMode(unsigned int mode, unsigned int WAIT);`
Returns	Nothing.
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. 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. 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);`

ECAN2GetOperationMode

Prototype	`unsigned int ECAN2GetOperationMode(void);`
Returns	Current operation mode.
Description	The function returns current operation mode of the ECAN2 module. Check the `ECAN_OP_MODE` constants (see ECAN constants) or device datasheet for operation mode codes.
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. Microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.

Example	`// check whether the ECAN2 module is in Normal mode and if it is`
	`// do something.`
	`if (ECAN2GetOperationMode() == ECAN_MODE_NORMAL)`
	`{`
	`  ...`
	`}`

ECAN2Initialize

Prototype	`void ECAN2Initialize(unsigned int SJW, unsigned int BRP, unsigned int PHSEG1, unsigned int PHSEG2, unsigned int PROPSEG, unsigned int ECAN_CONFIG_FLAGS);`
Returns	Nothing.
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 `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. Microcontroller must be connected to ECAN transceiver which is connected to the ECAN bus.

Example	`// initialize the ECAN2 module with appropriate baud rate and`
	`// message acceptance flags along with the sampling rules`
	`unsigned int ecan_config_flags;`
	`...`
	`ecan_config_flags = ECAN_CONFIG_SAMPLE_THRICE &`
	`// Form value to be used`
	`ECAN_CONFIG_PHSEG2_PRG_ON &`
	`// with ECANInitialize`
	`ECAN_CONFIG_XTD_MSG &`
	`ECAN_CONFIG_MATCH_MSG_TYPE &`
	`ECAN_CONFIG_LINE_FILTER_OFF;`
	
	`ECAN2Initialize(1, 3, 3, 3, 1, ecan_config_flags);`
	`// initialize the ECAN2 module`

ECAN2SelectTxBuffers

Prototype	`unsigned ECAN2SelectTxBuffers(unsigned txselect);`
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 claring 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. 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	`void ECAN2FilterDisable(unsigned fltdis);`
Returns	Nothing.
Description	The function disables receive filters. Parameters: - `fltdis`: filter disable selection parameter. Each bit corresponds to appropriate filter. By settung bit the corresponding filter is to be disabled.
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. 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	`void ECAN2FilterEnable(unsigned flten);`
Returns	Nothing.
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. 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	`unsigned ECAN2SetBufferSize(unsigned Ecan2BuffSize);`
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.h` header file located in the ECAN project folder
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. 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	`void ECAN2SetBaudRate(unsigned int SJW, unsigned int BRP, unsigned int PHSEG1, unsigned int PHSEG2, unsigned int PROPSEG, unsigned int ECAN_CONFIG_FLAGS);`
Returns	Nothing.
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. 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. 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`
	`unsigned int ecan_config_flags;`
	`...`
	`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 &`
	`// Form value to be used`
	`ECAN_CONFIG_PHSEG2_PRG_ON &`
	`// with ECAN2SetBaudRate`
	`ECAN_CONFIG_XTD_MSG &`
	`ECAN_CONFIG_MATCH_MSG_TYPE &`
	`ECAN_CONFIG_LINE_FILTER_OFF;`
	
	`ECAN2SetBaudRate(1, 3, 3, 3, 1, ecan_config_flags);`
	`// set the ECAN2 module baud rate`

ECAN2SetMask

Prototype	`void ECAN2SetMask(unsigned int ECAN_MASK, long val, unsigned int ECAN_CONFIG_FLAGS);`
Returns	Nothing.
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` & `ECAN_CONFIG_STD_MSG`, `ECAN_CONFIG_MATCH_MSG_TYPE` & `ECAN_CONFIG_XTD_MSG`. (see ECAN constants)
Requires	The ECAN2 routines are supported only by MCUs with the ECAN2 module. 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 &`
	`ECAN_CONFIG_XTD_MSG);`

ECAN2SetFilter

Prototype	`void ECAN2SetFilter(unsigned int ECAN_FILTER, long val, unsigned int ECAN_FILTER_MASK, unsigned int ECAN_FILTER_RXBUFF, unsigned int ECAN_CONFIG_FLAGS);`
Returns	Nothing.
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. 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	`unsigned int ECAN2Read(unsigned long *id, char *data, unsigned int *dataLen, unsigned int *ECAN_RX_MSG_FLAGS);`
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. 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.`
	`unsigned int msg_rcvd, rx_flags, data_len;`
	`char data[ 8] ;`
	`unsigned long msg_id;`
	`...`
	`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)) {`
	`  ...`
	`}`

ECAN2Write

Prototype	`unsigned int ECAN2Write(long id, char *Data, unsigned int DataLen, unsigned int ECAN_TX_MSG_FLAGS);`
Returns	`0` - if all Transmit Buffers are busy `0xFFFF` - if at least one Transmit Buffer is empty and available for transmition nsmit 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. 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`
	`unsigned int tx_flags;`
	`char data[ 8] ;`
	`unsigned long msg_id;`
	`...`
	`ECAN2SetOperationMode(ECAN_MODE_NORMAL,0xFF);`
	`// set NORMAL mode (ECAN2 must be in a mode in which transmis`
	`// sion is possible)`
	
	`tx_flags = ECAN_TX_PRIORITY_0 &`
	`ECAN_TX_XTD_FRAME &`
	`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 these as their argument:

```
const unsigned int
     ECAN_MODE_BITS      = 0x00E0,// Use this to access opmode  bits
     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 these (or a bitwise combination) as their argument:

```
const unsigned int
     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

     ECAN_CONFIG_MSG_TYPE_BIT     = 0x08,
     ECAN_CONFIG_STD_MSG          = 0xFF,    // XXXX1XXX
     ECAN_CONFIG_XTD_MSG          = 0xF7,    // XXXX0XXX

// continues..
```

```
// ..continued

    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 byte out of these values. For example:

```
init = ECAN_CONFIG_SAMPLE_THRICE &
       ECAN_CONFIG_PHSEG2_PRG_ON &
       ECAN_CONFIG_STD_MSG       &
       ECAN_CONFIG_MATCH_MSG_TYPE &
       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 these (or a bitwise combination) as their argument:

```
const unsigned int
    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 &
              ECAN_TX_XTD_FRAME  &
              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 or else it will be FALSE.

```
const unsigned int
    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 adjust the appropriate flags. For example:

```
if (MsgFlag & ECAN_RX_OVERFLOW != 0) {
  ...
  // Receiver overflow has occurred.
  // We have lost our previous message.
}
```

ECAN_MASK

The ECAN_MASK constants define mask codes. The functions ECAN1SetMask and ECAN2SetMask expect one of these as their argument:

```
const unsigned int
     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 these as their argument:

```
const unsigned int
     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

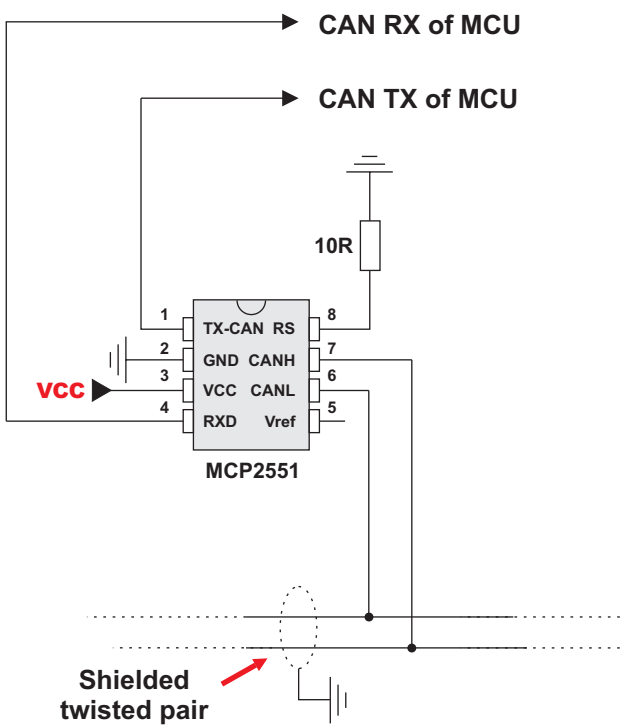
The code is a simple demonstration of the ECAN protocol.

```
#include "ECan_Defs.h"

unsigned int Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags;  // can flags
unsigned int Rx_Data_Len;                                    // received data length in bytes
char RxTx_Data[ 8] ;                                         // can rx/tx data buffer
char Msg_Rcvd;                                               // reception flag
unsigned long Tx_ID, Rx_ID;                                  // can rx and tx ID

void C1Interrupt(void) org 0x005A                            // ECAN event iterrupt
{

       IFS2bits.C1IF = 0;                                    // clear ECAN interrupt flag
       if(C1INTFbits.TBIF) {                                 // was it tx interrupt?
       C1INTFbits.TBIF = 0;                                  // if yes clear tx interrupt flag
  }

  if(C1INTFbits.RBIF) {                                      // was it rx interrupt?
             C1INTFbits.RBIF = 0;                            // if yes clear rx interrupt flag
      }
}

void main() {

  // Set PLL : Fosc = ((Fin/PLLPRE)*PLLDIV)/PLLPOST ; (((10MHz/2)*32)/4) = 20MHz
  // refer the pic24 family datasheet for more details
  CLKDIV &= 0xFFE0;  //CLKDIVbits.PLLPRE = 0;
  PLLFBD  = 0x1E;    //PLLFBDbits.PLLDIV = 0x1E;
  CLKDIV &= 0xFF3F;  //CLKDIVbits.PLLPOST = 1;
  CLKDIV |= 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 */
      IEC2bits.C1IE    = 1;              // enable ECAN1 interrupts
      C1INTEbits.TBIE = 1;               // enable ECAN1 tx interrupt
      C1INTEbits.RBIE = 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 &             // Form value to be used
                   ECAN_TX_XTD_FRAME &              //  with CANSendMessage
                   ECAN_TX_NO_RTR_FRAME;

  Can_Init_Flags = ECAN_CONFIG_SAMPLE_THRICE &      // Form value to be used
                   ECAN_CONFIG_PHSEG2_PRG_ON &      //  with CANInitialize
                   ECAN_CONFIG_XTD_MSG &
                   ECAN_CONFIG_MATCH_MSG_TYPE &
                   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

  ECAN1SetMask(ECAN_MASK_0, -1, ECAN_CONFIG_MATCH_MSG_TYPE &
ECAN_CONFIG_XTD_MSG);        // set all mask1 bits to ones
  ECAN1SetMask(ECAN_MASK_1, -1, ECAN_CONFIG_MATCH_MSG_TYPE &
ECAN_CONFIG_XTD_MSG);        // set all mask2 bits to ones
  ECAN1SetMask(ECAN_MASK_2, -1, ECAN_CONFIG_MATCH_MSG_TYPE &
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
//continues...
```

```
// continued ...
  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 (1) { // endless loop
    Msg_Rcvd = ECAN1Read(&Rx_ID , RxTx_Data , &Rx_Data_Len, &Can_Rcv_Flags);
    // read received message
    if ((Rx_ID == 3u) && Msg_Rcvd) {
    // if message received check id
      PORTB = RxTx_Data[ 0] ;
    // id correct, output data at PORTB
      RxTx_Data[ 0] ++ ;
    // increment received data
      Delay_ms(10);
      ECAN1Write(Tx_ID, RxTx_Data, 1, Can_Send_Flags);
    // send incremented data back
    }
  }
} //~!
```

Hardware Connection



EEPROM Library

EEPROM data memory is available with a number of dsPIC30 family. The mikroC for dsPIC30/33 and PIC24 includes a library for comfortable work with MCU's internal EEPROM.

Note: Only dsPIC30 MCUs have EEPROM memory.

Library Routines

```
Eeprom_Erase
Eeprom_Erase_Block
Eeprom_Read
Eeprom_Write
Eeprom_Write_Block
```

Eeprom_Erase

Prototype	`void Eeprom_Erase(unsigned long address);`
Returns	Nothing.
Description	Erases a single (16-bit) location from EEPROM memory. Parameters : - `address`: address of the EEPROM memory 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.
Requires	Nothing.

Example	`unsigned long eeAddr = 0x7FFC80;`
	`...`
	`Eeprom_Erase(eeAddr);`

Eeprom_Erase_Block

Prototype	`void Eeprom_Erase_Block(unsigned long address);`
Returns	Nothing.
Description	Erases one EEPROM row (16 words block) from EEPROM memory. Parameters : - `address`: starting address of the EEPROM memory 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.
Requires	Nothing.

Example	`unsigned long eeAddr = 0x7FFC20;`
	`...`
	`Eeprom_Erase_Block(eeAddr);`

Eeprom_Read

Prototype	`unsigned int Eeprom_Read(unsigned long address);`
Returns	Word from the specified address.
Description	Reads data from specified `address`. Parameters : - `address`: address of the EEPROM memory location to be read.
Requires	It is the user's responsibility to obtain proper address parity (in this case, even).

Example	`unsigned long eeAddr = 0x7FFC20;`
	`unsigned int temp;`
	`...`
	`temp = Eeprom_Read(eeAddr);`

Eeprom_Write

Prototype	`void Eeprom_Write(unsigned long address, unsigned int data);`
Returns	Nothing.
Description	Writes data to specified address. Parameters : address: address of the EEPROM memory location to be written. data: data to be written. **Note**: Specified memory location will be erased before writing starts.
Requires	Nothing.

Example	`unsigned int eeWrite = 0xAAAA;`
	`unsigned long wrAddr = 0x7FFC30;`
	`...`
	`Eeprom_Write(wrAddr, eeRead);`

Eeprom_Write_Block

Prototype	`void Eeprom_Write_Block(unsigned long address, unsigned int *data);`
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	`unsigned long wrAddr = 0x7FFC20;`
	`unsigned int iArr[16] = {'m', 'i', 'k', 'r', 'o', 'E', 'l', 'e', 'k', 0};`
	`...`
	`Eeprom_Write_Block(wrAddr, iArr);`

Library Example

The example demonstrates using the EEPROM Library. Hardware configurations in this example are made for the dsPICPRO2 board and dsPIC30F6014A.

```
#include <built_in.h>

unsigned int eeRead;
unsigned long eeAddr;

unsigned int iArr[16] = {'m', 'i', 'k', 'r', 'o', 'E', 'l', 'e', 'k', 0};
unsigned char dArr[16];

void main() {

  unsigned i;

  ADPCFG = 0xFFFF;    //--- disable analog inputs
  TRISB = 0;
  LATB = 0xFFFF;
  eeRead = 0xAAAA;
  eeAddr = 0x7FFC80;

  while (eeRead < 0xAABA) {
    Eeprom_Write(eeAddr, eeRead++);
    while(NVMCONbits.WR) ;                    // wait for write to finish,
    LATB = Eeprom_Read(eeAddr);               // then, read the just-written
    eeAddr += 2;                              // data.

    Delay_ms(500);
  }

  //--- write entire row of EEPROM data,...
  Eeprom_Write_Block(0x7FFC20, iArr);
  while(NVMCONbits.WR) ;

  //--- ...read the data back...
  eeRead = 1; eeAddr = 0x7FFC20; i = 0;
  while(eeRead) {
    eeRead = Eeprom_Read(eeAddr);
    dArr[i++] = Lo(eeRead);
    eeAddr += 2;
  }

  //--- ...and display it on LCD.
  Lcd_Custom_Config(&PORTD, 7,6,5,4, &PORTB, 4,0,6);
  Lcd_Custom_Out(2,1, dArr);

} //~!
```

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 mikroC 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 mikroC 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	`void Flash_Erase32(unsigned long address);`
Description	Erases one block (32 instructions, 64 addresses, 96 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 32-instruction block, starting from address`
	`// 0x006000`
	`Flash_Erase32(0x006000);`

Flash_Write_Block

Prototype	`void Flash_Write_Block(unsigned long address, unsigned int *data);`
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 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 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	`unsigned long flash_address = 0x006000;`
	`unsigned int Buffer[4] = {'A', 'B', 'C', 'D'};`
	`...`
	`Flash_Write_Block(flash_address, Buffer);`

Flash_Write_Compact

Prototype	`void Flash_Write_Compact(unsigned long address, void *data, unsigned bytes);`
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 : - `address`: starting address of the FLASH memory block - `data`: data 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	`unsigned long flash_address = 0x006000;`
	`char cArr[] = "supercalifragillisticexpialidotious";`
	`...`
	`Flash_Write_Compact(flash_address, Buffer, 36);`

Flash_Write_Init

Prototype	`void Flash_Write_Init(unsigned long address, void* data);`
Description	Initializes RTSP for write-to-FLASH operation. 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_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	`//--- Initializes the Flash to be written, starting from`
	`// address 0x006100, the data is located at *pv1`
	`void *pv1;`
	`...`
	`Flash_Write_Init(0x006100, pv1);`

Flash_Write_Loadlatch4

Prototype	`void Flash_Write_Loadlatch4(void);`
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 the 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	`//--- writes data from an array, in "regular" manner`
	`unsigned int iArr[16] = {'m', 'i', 'k', 'r', 'o', 'E', 'l', 'e', 'k'};`
	`void * pv1;`
	`...`
	`pv1 = iArr;`
	`Flash_Write_Init(0x006100, pv1);`
	`Flash_Write_Loadlatch4();`
	`Flash_Write_Loadlatch4();`
	`Flash_Write_DoWrite();`

Flash_Write_Loadlatch4_Compact

Prototype	`void Flash_Write_Loadlatch4_Compact(void);`
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 the 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	`//--- writes data from an array of char, in "compact" manner`
	`char cArr[] = "supercalifragillisticexpialidotious"; //35+1 bytes`
	`void * pv1;`
	`...`
	`pv1 = cArr;`
	`Flash_Write_Init(0x006000, pv1);  //init`
	`Flash_Write_Loadlatch4_Compact(); //12 bytes`
	`Flash_Write_Loadlatch4_Compact(); //12 bytes`
	`Flash_Write_Loadlatch4_Compact(); //12 bytes`
	`Flash_Write_DoWrite();  //commit write`

Flash_Write_DoWrite

Prototype	`void Flash_Write_DoWrite(void);`
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 the 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 and of the FLASH write sequence.

Example	`//--- writes data from an array, in "regular" manner`
	`unsigned int iArr[16] = {'m', 'i', 'k', 'r', 'o', 'E', 'l', 'e', 'k'};`
	`void * pv1;`
	`...`
	`pv1 = iArr;`
	`Flash_Write_Init(0x006100, pv1);`
	`Flash_Write_Loadlatch4();`
	`Flash_Write_Loadlatch4();`
	`Flash_Write_DoWrite();`

Flash_Read4

Prototype	`unsigned int* Flash_Read4(unsigned long address, unsigned int *write_to);`
Returns	Starting address of RAM buffer for storing read data.
Description	Reads one latch row (4 instructions, 8 addresses) 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
Requires	**Note**: The user should take care of the address alignment (see the explanation at the beginning of this page).

Example	`//--- reads 8 bytes (4 words) from location 0x006000 and stores`
	`// it to *pv1;`
	`unsigned int *pv1;`
	`...`
	`Flash_Read4(0x006000, pv1);`

Flash_Read4_Compact

Prototype	`void* Flash_Read4_Compact(unsigned long address, void *write_to);`
Returns	Starting address of RAM buffer for storing read data.
Description	Reads one latch row (4 instructions, 8 addresses) 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
Requires	**Note**: The user should take care of the address alignment (see the explanation at the beginning of this page).

Example	`//--- reads 12 bytes (4 words) from location 0x006000 and stores`
	`// it to *pv1;`
	`unsigned int *pv1;`
	`...`
	`Flash_Read4_Compact(0x006000, pv1);`

PIC24 and dsPIC33:

Flash_Erase

Prototype	`void Flash_Erase(unsigned long address);`
Returns	Nothing.
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`
	`unsigned long flash_address = 0x006400;`
	`...`
	`Flash_Erase(flash_address);`

Flash_Write

Prototype	`void Flash_Write(unsigned long address, unsigned int *data);`
Returns	Nothing.
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 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 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	`unsigned int iArr[64] = {'m', 'i', 'k', 'r', 'o', 'E', 'l', 'e', 'k', 't', 'r', 'o', 'n', 'i', 'k', 'a'};`
	`void * pv1;`
	`...`
	`pv1 = iArr;`
	`Flash_Write(0x006500, pv1);`

Flash_Write_Compact

Prototype	`void Flash_Write_Compact(unsigned long address, char *data);`
Returns	Nothing.
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 the 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	`char cArr[] = "supercalifragillisticexpialidotiousABCDEFGHIJKLMNOPRSTUVWXYZ1234";`
	`void * pv1;`
	`...`
	`pv1 = cArr;`
	`Flash_Write_Compact(0x006400, pv1);`

Flash_Read

Prototype	`unsigned int* Flash_Read(unsigned long address, unsigned int *write_to, unsigned NoWords);`
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	`unsigned Buffer[ 64] ;`
	`unsigned long start_address = 0x6500;`
	`...`
	`Flash_Read(start_address, Buffer, 10);`

Flash_Read_Compact

Prototype	`void* Flash_Read_Compact(unsigned long address, void *write_to, unsigned NoBytes);`
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	`char Buffer[ 64] ;`
	`unsigned long start_address = 0x6500;`
	`...`
	`Flash_Read_Compact(start_address, Buffer, 10);`

Library Example

In these examples, various techniques of reading/writing from/to on-chip FLASH memory are shown.

dsPIC30:

```
char cArr[] = "supercalifragillisticexpialidotious";
char cArr2[40];
void * pv1;
unsigned bb;

void main() {
  unsigned i;

  pv1 = cArr;

/*
  // This is what Flash_Write_Compact() does 'benath the hood'
  Flash_Write_Init(0x006000, pv1);
  Flash_Write_Loadlatch4_Compact();
  Flash_Write_Loadlatch4_Compact();
  Flash_Write_Loadlatch4_Compact();
  Flash_Write_DoWrite();
 */

  //--- 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 += 12;
  Flash_Read4_Compact(0x006008, pv1);
  pv1 += 12;
  Flash_Read4_Compact(0x006010, pv1);
  pv1 += 12;
  *pv1 = 0;  //termination

  //--- show what has been written
  i = 0;
  Uart1_Init(9600);
  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(cArr2[ i] ) {
      bb = cArr2[ i++] ;
      Uart1_Write_Char(bb);
    }

    //--- now for some non-compact flash-write
    pv1 = iArr;
    //--- erase the block first
    Flash_Erase32(0x006100);
    Flash_Write_Init(0x006100, pv1);
    Flash_Write_Loadlatch4();
    Flash_Write_Loadlatch4();
    Flash_Write_DoWrite();
  } //~!
```

PIC24 and dsPIC33:

```
unsigned int iArr[ 64] = { 'm', 'i', 'k', 'r', 'o', 'E', 'l', 'e', 'k', 't', 'r',
'o', 'n', 'i', 'k', 'a'} ;
char cArr[ ] = "supercalifragillisticexpialidotiousABCDEFGHIJKLMNOPRSTUVWXYZ1234";
char cArr2[ 64] ;
void * pv1;

void main() {
  unsigned i, k;

  AD1PCFG = 0xFFFF;
  PORTB = 0x0004;
  TRISB = 1;

  for(i=16;i<64;i++)
    iArr[ i] = 'x';

  for(i=64;i<192;i++)
    cArr[ i] = 'y';

  // press RB0 to continue
  while (!PORTB.f0)
    ;
  //--- erase the block first
  Flash_Erase(0x006400);

  //--- now for some non-compact flash-write
  pv1 = iArr;
  Flash_Write(0x006500, pv1);

  PORTB = 0x0008;

//continues ...
```

```
//continued...

  //--- write compact format to flash
  pv1 = cArr;
  Flash_Write_Compact(0x006400, pv1);

  PORTB = 0x0010;

  //--- read compact format
  pv1 = cArr2;
  Flash_Read_Compact(0x006400, pv1, 64);
  pv1[ 64] = 0;  //termination

  //--- show what has been written
  i = 0;
  Uart1_Init(9600);
  Uart1_Write_Char('s'); Uart1_Write_Char('t'); Uart1_Write_Char('a');
  Uart1_Write_Char('r');
  Uart1_Write_Char('t'); Uart1_Write_Char(10);
  while(cArr2[ i] )
    Uart1_Write_Char(cArr2[ i++] );

  PORTB = 0x0018;

} //~!
```

Graphic LCD Library

The mikroC for dsPIC30/33 and PIC24 provides a library for drawing and writing on the commonly used Samsung (KS108/KS107) Graphic LCD 128x64.

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	`void Glcd_Init(unsigned int *cs1_port, unsigned int cs1_pin, unsigned int *cs2_port, unsigned int cs2_pin, unsigned int *rs_port, unsigned int rs_pin, unsigned int *rw_port, unsigned int rw_pin, unsigned int *rst_port, unsigned int rst_pin, unsigned int *en_port, unsigned int en_pin, unsigned int *data_port);`
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	`void Glcd_Config(unsigned *cs1_port, unsigned cs1_pin, unsigned *cs2_port, unsigned cs2_pin, unsigned *rs_port, unsigned rs_pin, unsigned *rw_port, unsigned rw_pin, unsigned *rst_port, unsigned rst_pin, unsigned *en_port, unsigned en_pin, unsigned *data_portLo, unsigned *data_portHi, char ports_config);`

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 are determining 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 rou`
	`// tine`
	`Glcd_Init_EasyDsPIC3();`

Glcd_Set_Side

Prototype	`void Glcd_Set_Side(unsigned short x_pos);`
Description	Selects GLCD side. Refer to the GLCD datasheet for detailed explaination. Parameters : `x_pos`: position on x-axis. Valid values: 0..127 The parameter `x_pos` specifies the GLCD side: values from 0 to 63 specify the left side, values from 64 to 127 specify the right side. **Note**: For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see Glcd_Init 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	`void Glcd_Set_X(unsigned short x_pos);`
Description	Sets x-axis position to x_pos dots from the left border of GLCD within the selected side. Parameters : `x_pos`: position on x-axis. Valid values: 0..63 **Note**: For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see `Glcd_Init` and `Glcd_Config` routines.
Example	`Glcd_Set_X(25);`

Glcd_Set_Page

Prototype	`void Glcd_Set_Page(unsigned short page);`
Returns	Nothing.
Description	Selects page of the GLCD. Parameters : `page`: page number. Valid values: `0..7` **Note**: For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see `Glcd_Init` and `Glcd_Config` routines.
Example	`Glcd_Set_Page(5);`

Glcd_Read_Data

Prototype	`unsigned int Glcd_Read_Data(void);`
Returns	One byte from GLCD memory, formatted as a word (16-bit).
Description	Reads data from from the current location of GLCD memory and moves to the next location.
Requires	GLCD needs to be initialized, see `Glcd_Init` 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	`unsigned int data;`
	`...`
	`data = Glcd_Read_Data();`

Glcd_Write_Data

Prototype	`void Glcd_Write_Data(unsigned short ddata);`
Description	Writes one byte to the current location in GLCD memory and moves to the next location. Parameters : `ddata`: data to be written.
Requires	GLCD needs to be initialized, see `Glcd_Init` 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	`unsigned short data;`
	`...`
	`Glcd_Write_Data(data);`

Glcd_Fill

Prototype	`void Glcd_Fill(unsigned short pattern);`
Description	Fills GLCD memory with the byte pattern.Parameters : - `pattern`: byte to fill GLCD memory with To clear the GLCD screen, use `Glcd_Fill(0)`. **To fill the screen completely, use** `Glcd_Fill(0xFF)`**.**
Requires	GLCD needs to be initialized, see `Glcd_Init` and `Glcd_Config` routines.
Example	`// Clear screen` `Glcd_Fill(0);`

Glcd_Dot

Prototype	`void Glcd_Dot(unsigned short x_pos, unsigned short y_pos, unsigned short color);`
Description	Draws a dot on GLCD at coordinates (x_pos, y_pos). Parameters : x_pos: x position. Valid values: 0..127 y_pos: y position. Valid values: 0..63 color: color parameter. Valid values: 0..2 The parameter color determines a dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state. **Note**: For x and y axis layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see `Glcd_Init` and `Glcd_Config` routines.
Example	`// Invert the dot in the upper left corner` `Glcd_Dot(0, 0, 2);`

Glcd_Line

Prototype	`void Glcd_Line(int x_start, int y_start, int x_end, int y_end, unsigned short color);`
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	`void Glcd_V_Line(unsigned short y_start, unsigned short y_end, unsigned short x_pos, unsigned short color);`
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	`void Glcd_H_Line(unsigned short x_start, unsigned short x_end, unsigned short y_pos, unsigned short color);`
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	`void Glcd_Rectangle(unsigned short x_upper_left, unsigned short y_upper_left, unsigned short x_bottom_right, unsigned short y_bottom_right, unsigned short color);`
Description	Draws a rectangle on GLCD. Parameters : - `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right rectangle corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized, see `Glcd_Init` 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	`void Glcd_Box(unsigned short x_upper_left, unsigned short y_upper_left, unsigned short x_bottom_right, unsigned short y_bottom_right, unsigned short color);`
Description	Draws a box on GLCD. Parameters : - `x_upper_left`: x coordinate of the upper left box corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left box corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right box corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right box corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the box fill: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized, see `Glcd_Init` 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	`void Glcd_Circle(int x_center, int y_center, int radius, unsigned short color);`
Description	Draws a circle on GLCD. Parameters : - `x_center`: x coordinate of the circle center. Valid values: 0..127 - `y_center`: y coordinate of the circle center. Valid values: 0..63 radius: radius size color: color parameter. Valid values: 0..2 The parameter color determines the color of the circle line: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized, see `Glcd_Init` 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	`void Glcd_Set_Font(const char *activeFont, unsigned short aFontWidth, unsigned short aFontHeight, unsigned int aFontOffs);`
Description	Sets font that will be used with `Glcd_Write_Char` and `Glcd_Write_Text` routines. Parameters : - `activeFont`: font to be set. Needs to be formatted as an array of char - `aFontWidth`: width of the font characters in dots. - `aFontHeight`: height of the font characters in dots. - `aFontOffs`: number that represents difference between the mikroC for dsPIC30/33 and PIC24 character set and regular ASCII set (eg. if 'A' is 65 in ASCII character, and 'A' is 45 in the mikroC 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.c`" file located in the Uses folder or create his own fonts.
Requires	GLCD needs to be initialized, see `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	`void Glcd_Write_Char(unsigned short chr, unsigned short x_pos, unsigned short page_num, unsigned short color);`
Description	Prints character on the GLCD. Parameters : - `chr`: character to be written - `x_pos`: character starting position on x-axis. Valid values: 0..(127-FontWidth) - `page_num`: the number of the page on which character will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the character: 0 white, 1 black, and 2 inverts each dot. **Note**: For x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see `Glcd_Init` 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	`void Glcd_Write_Text(char *text, unsigned short x_pos, unsigned short page_num, unsigned short color);`
Description	Prints text on GLCD. Parameters : - `text`: text to be written - `x_pos`: text starting position on x-axis. - `page_num`: the number of the page on which text will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the text: 0 white, 1 black, and 2 inverts each dot. **Note**: For x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized, see `Glcd_Init` 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	`void Glcd_Image(const unsigned short *image);`
Description	Displays bitmap on GLCD. Parameters : - `image`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroC for dsPIC30/33 and PIC24 pointer to const and pointer to RAM equivalency). Use the mikroC'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 `Glcd_Init` and `Glcd_Config` routines.
Example	`// Draw image my_image on GLCD` `Glcd_Image(my_image);`

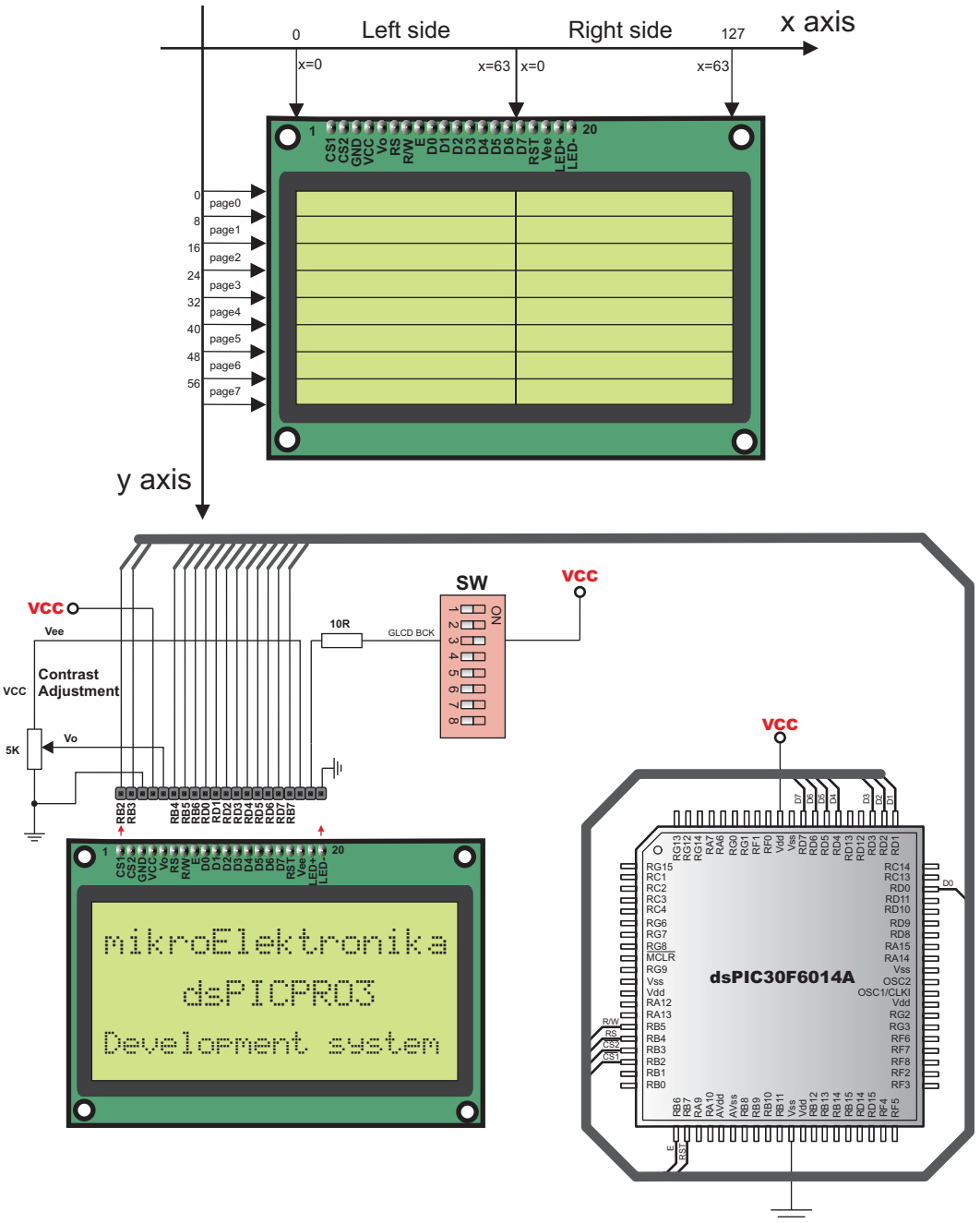
Library Example

The following drawing demo tests advanced routines of GLCD library.

```
#include "bmp1.h"

char cArr[20];
char *someText;

void Delay2S(){
  delay_ms(2000);
}//~

void main() {
  unsigned short ii;
  unsigned int jj;

  sometext = cArr;

  //--- turn off A/D inputs
  ADPCFG = 0xFFFF;
  // Init for dsPICPRO3 development system
  Glcd_Init(&PORTB,2, &PORTB,3, &PORTB,4, &PORTB,5, &PORTB,7, &PORTB,6, &PORTD);
  Delay_100ms();

// continues...
```

```
  //continued...

  lMainLoop:

    Glcd_Fill(0x00);
    Glcd_Image( maska_bmp );
    Delay2S();

    Glcd_Fill(0x00);
    Glcd_Circle(63,32, 20, 1);
    Delay2S();
    Glcd_Line(120,1, 5,60, 1);
    Glcd_Line(12,42, 5,60, 1);
    Delay2S();

    Glcd_Rectangle(12,20, 93,57, 1);
    Delay2S();

    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();

    for (ii = 1; ii <= 10; ii++)
      Glcd_Circle(63,32, 3*ii, 1);
    Delay2S();

    Glcd_Box(12,20, 70,57, 2);
    Delay2S();

    Glcd_Set_Font(defaultFont, 5,7, 48);
    someText = "BIG:ONE";
    Glcd_Write_Text(someText, 5,3, 2);
    Delay2S();


    someText = "SMALL:NOT:SMALLER";
    Glcd_Write_Text(someText, 20,5, 1);
    Delay2S();

    Glcd_Fill(0x00);
    Glcd_Set_Font(System3x6, 3, 6, 0x20);
    Glcd_Write_Text(someText, 10,5, 1);
    Delay2S();

    goto lMainLoop;
  } //~!
```

Hardware Connection



I2C Library

The I²C full master MSSP module is available with a number of the dsPIC30/33 and PIC24 MCU models. The mikroC 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 (supports both I²C modules) libraries. Switching between I²C modules in I2C library is done by the I2C_Set_Active function (both I²C modules have to be previously initialized).

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
I2C_Set_Active
```

I2C_Init

Prototype	`void I2C_Init(unsigned long scl);`
Description	Configures and initializes the I²C (dsPIC30) i.e. I²C1 (dsPIC33 and PIC24) module with default settings. 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 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 the 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	`void I2C_Start(void);`
Returns	Nothing.
Description	Determines if the I²C bus is free and issues START signal. **Note**: if MCU has 2 I²C modules, active module will be used. See `I2C_Set_Active` routine.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See `I2C1_Init`, `I2C2_Init` and `I2C_Init` routines.
Example	`// Issue START signal` `I2C_Start();`

I2C_Restart

Prototype	`void I2C_Restart(void);`
Description	Issues repeated START signal. **Note**: if MCU has 2 I²C modules, active module will be used. See I2C_Set_Active routine.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See `I2C1_Init`, `I2C2_Init` and `I2C_Init` routines.
Example	`// Issue RESTART signal` `I2C_Restart();`

I2C_Wait_For_Idle

Prototype	`void I2C_Wait_For_Idle(void);`
Returns	Nothing.
Description	Waits for the I²C bus to become free. This is a blocking function. **Note**: if MCU has 2 I²C modules, active module will be used. See I2C_Set_Active routine.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See `I2C1_Init`, `I2C2_Init` and `I2C_Init` routines.

Example	`unsigned char data;`
	`...`
	`I2C_Wait_For_Idle();`
	`I2C_Write(data);`
	`...`

I2C_Read

Prototype	`unsigned char I2C_Read(unsigned ack);`
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. **Note**: if MCU has 2 I²C modules, active module will be used. See I2C_Set_Active routine.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See `I2C1_Init`, `I2C2_Init` and `I2C_Init` routines. Also, START signal needs to be issued in order to use this function. See `I2C_Start`.

Example	`unsigned char take;`
	`...`
	`// Read data and send the not_acknowledge signal`
	`take = I2C_Read(1);`

I2C_Write

Prototype	`unsigned I2C_Write(unsigned char data);`
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 **Note**: if MCU has 2 I²C modules, active module will be used. See I2C_Set_Active routine.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See `I2C1_Init`, `I2C2_Init` and `I2C_Init` routines. Also, START signal needs to be issued in order to use this function. See `I2C_Start`.

Example	`unsigned char data;`
	`unsigned error;`
	`...`
	`error = I2C_Write(data);`
	`error = I2C_Write(0xA3);`

I2C_Stop

Prototype	`void I2C_Stop(void);`
Description	Issues STOP signal. **Note**: if MCU has 2 I²C modules, active module will be used. See I2C_Set_Active routine.
Requires	MCU with at least one I²C module. Used I²C module must be initialized before using this function. See `I2C1_Init`, `I2C2_Init` and `I2C_Init` routines.
Example	`// Issue STOP signal` `I2C_Stop();`

I2c_Set_Active

Prototype	`void I2C_Set_Active(char I2cNo);`
Returns	Nothing.
Description	Sets the active I²C module which will be used by `I2C_Start`, `I2C_Restart`, `I2C_Wait_For_Idle`, `I2C_Read`, `I2C_Write` and `I2C_Stop` routines. Parameters : `I2cNo`: module number. Valid values: 1 (I²C1) and 2 (I²C2).
Requires	Routine is available only for MCUs with two I²C modules. Used I²C module must be initialized before using this function. See `I2C1_Init`, `I2C2_Init` and `I2C_Init` routines.
Example	`//Set I2C2 module active` `I2C_Set_Active(2);`

I2C1_Init

Prototype	`void I2C1_Init(unsigned long scl);`
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 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	`void I2C1_Start(void);`
Returns	Nothing.
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	`void I2C1_Restart(void);`
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	`void I2C1_Wait_For_Idle(void);`
Returns	Nothing.
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	`unsigned char data;`
	`...`
	`I2C1_Wait_For_Idle();`
	`I2C1_Write(data);`
	`...`

I2C1_Read

Prototype	`unsigned char I2C1_Read(unsigned ack);`
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	`unsigned char take;`
	`...`
	`// Read data and send the not_acknowledge signal`
	`take = I2C1_Read(1);`

I2C1_Write

Prototype	`unsigned I2C1_Write(unsigned char data);`
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	`unsigned char data;`
	`unsigned error;`
	`...`
	`error = I2C1_Write(data);`
	`error = I2C1_Write(0xA3);`

I2C1_Stop

Prototype	`void I2C1_Stop(void);`
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.
Examples	`// Issue STOP signal` `I2C1_Stop();`

I2C2_Init

Prototype	`void I2C2_Init(unsigned long scl);`
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 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	`void I2C2_Start(void);`
Returns	Nothing.
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	`void I2C2_Restart(void);`
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	`void I2C2_Wait_For_Idle(void);`
Returns	Waits for the I²C bus to become free. This is a blocking function.
Description	Waits for I²C bus to become free.
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	`unsigned char data;`
	`...`
	`I2C2_Wait_For_Idle();`
	`I2C2_Write(data);`
	`...`

I2C2_Read

Prototype	`unsigned char I2C2_Read(unsigned ack);`
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	`unsigned char take;`
	`...`
	`// Read data and send the not_acknowledge signal`
	`take = I2C2_Read(1);`

I2C2_Write

Prototype	`unsigned I2C2_Write(unsigned char data);`
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	`unsigned char data;`
	`unsigned error;`
	`...`
	`error = I2C2_Write(data);`
	`error = I2C2_Write(0xA3);`

I2C2_Stop

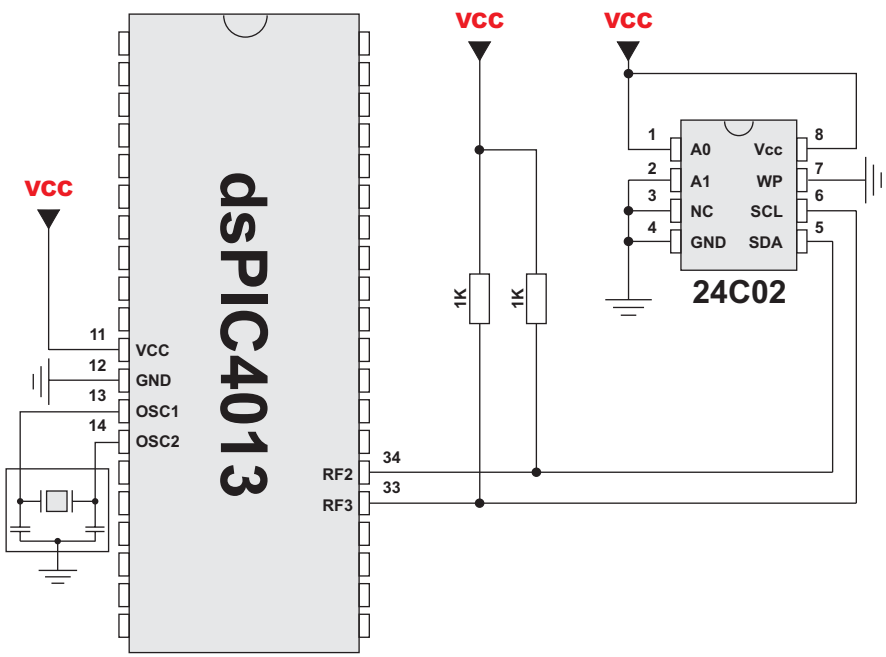
Prototype	`void I2C2_Stop(void);`
Description	Issues STOP signal.
Requires	MCU with the I²C2 module. The I²C2 module must be initialized before using this function. See I2C2_Init. Supported by the dsPIC33 and PIC24 MCUs only.
Example	`// Issue STOP signal` `I2C2_Stop();`

Library Example

This code demonstrates working with the I²C library. Program sends data to EEPROM (data is written at the address 2). After that, program reads data from the same EEPROM address and displays it on PORTB for visual check. See the figure below how to interface the 24C02 to dsPIC30/33 and PIC24.

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

  dAddr = 0x02;

  I2c_Init(100000);
  I2c_Start();                    // issue I2C start signal
  I2c_Write(0xA2);                // send byte via I2C  (command to 24cO2)
  I2c_Write(dAddr);               // send byte (address of EEPROM location)
  I2c_Write(0xF4);                // send data (data to be written)
  I2c_Stop();

  Delay_100ms();

  I2c_Start();                    // issue I2C start signal
  I2c_Write(0xA2);                // send byte via I2C  (device address + W)
  I2c_Write(0x02);                // 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 (NO acknowledge)
  I2c_Stop();

} //~!
```

Hardware Connection



KEYPAD LIBRARY

The mikroC 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 connections explanation see schematic at the bottom of this page.

Library Routines

```
Keypad_Init
Keypad_Key_Press
Keypad_Key_Click
```

Keypad_Init

Prototype	`void Keypad_Init(unsigned *key_port);`
Description	Initializes given port for working with keypad. Parameters : `key_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	`unsigned Keypad_Key_Press(void);`
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	`unsigned kp;`
	`...`
	`kp = Keypad_Key_Press();`

Keypad_Key_Click

Prototype	`unsigned Keypad_Key_Click(void);`
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	`unsigned kp;`
	`...`
	`kp = Keypad_Key_Click();`

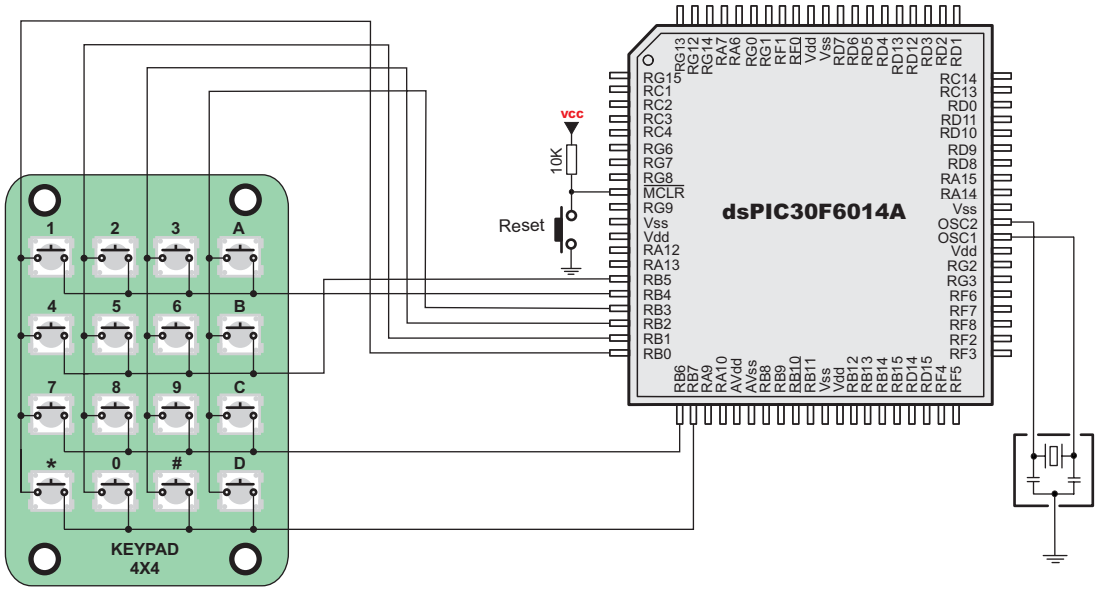
Library Example

The following code can be used for testing the keypad. It is written for keypad_4x3 or _4x4. The code returned by the keypad functions (1..16) is transformed into ASCII codes [0..9,A..F], and is then sent via UART1.

```
unsigned kp;

void main() {

  ADPCFG = 0xFFFF;

  Keypad_Init(&PORTB);    // PORTB [7..0]
  Uart1_Init(9600);
  Delay_ms(200);
  Uart1_Write_Char('R');

  do {
    //--- Wait for key to be pressed
    do {
      //kp = Keypad_Key_Click();      // choose the key detecting function
      kp = Keypad_Key_Press();
    } while (!kp);



// continues...
```

```
    // continued...


    //--- Prepare value for output
    switch (kp) {


    // uncomment this block for keypad4x3 //

      /* case 10: kp = 42; break;  // '*'
         case 11: kp = 48; break;  // '0'
         case 12: kp = 35; break;  // '#'
         default: kp += 48;                        */



    // uncomment this block for keypad4x4 //

      case  1: kp = 49; break; // 1
      case  2: kp = 50; break; // 2
      case  3: kp = 51; break; // 3
      case  4: kp = 65; break; // A
      case  5: kp = 52; break; // 4
      case  6: kp = 53; break; // 5
      case  7: kp = 54; break; // 6
      case  8: kp = 66; break; // B
      case  9: kp = 55; break; // 7
      case 10: kp = 56; break; // 8
      case 11: kp = 57; break; // 9
      case 12: kp = 67; break; // C
      case 13: kp = 42; break; // *
      case 14: kp = 48; break; // 0
      case 15: kp = 35; break; // #
      case 16: kp = 68; break; // D

    }

    //--- Send on UART1
    Uart1_Write_Char(kp);

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

Hardware Connection



LCD Custom Library (4-bit interface)

The mikroC 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.

Note: mikroElektronika's development system based initialization routines can be found in the setup library files located in the Uses folder.

Note: Only `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_Custom_Config
Lcd_Custom_Out
Lcd_Custom_Out_Cp
Lcd_Custom_Chr
Lcd_Custom_Chr_Cp
Lcd_Custom_Cmd
```

Lcd_Custom_Config

Prototype	`void Lcd_Custom_Config(unsigned *data_port, char db3, char db2, char db1, char db0, unsigned *ctrl_port, char rs, char ctrl_rw, char enable);`
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_Custom_Config(&PORTB, 3,2,1,0, &PORTD, 0,2,1);`
	
	`// or just call the appropriate system based initialization rou`
	`// tine`
	`Lcd_Custom_Init_EasyDsPIC3();`

Lcd_Custom_Out

Prototype	`void Lcd_Custom_Out(char row, char column, char *text);`
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	Port with LCD must be initialized. See `Lcd_Config`.
Example	`// Write text "Hello!" on LCD starting from row 1, column 3:` `Lcd_Custom_Out(1, 3, "Hello!");`

Lcd_Custom_Out_Cp

Prototype	`void Lcd_Custom_Out_Cp(char *text);`
Description	Prints text on LCD at current cursor position. Both string variables and literals can be passed as a text. Parameters : - `text`: text to be written
Requires	The LCD module needs to be initialized. See `Lcd_Custom_Config` routine.
Example	`// Write text "Here!" at current cursor position:` `Lcd_Custom_Out_Cp("Here!");`

Lcd_Custom_Chr

Prototype	`void Lcd_Custom_Chr(unsigned row, unsigned column, char out_char);`
Description	Prints character on LCD at specified position. Both variables and literals can be passed as a character. Parameters : - `row`: writing position row number - `column`: writing position column number - `out_char`: character to be written
Requires	The LCD module needs to be initialized. See `Lcd_Custom_Config` routine.
Example	`// Write character "i" at row 2, column 3:` `Lcd_Custom_Chr(2, 3, 'i');`

Lcd_Custom_Chr_Cp

Prototype	`void Lcd_Custom_Chr_Cp(char out_char);`
Description	Prints character on LCD at current cursor position. Both variables and literals can be passed as a character. Parameters : `out_char`: character to be written
Requires	The LCD module needs to be initialized. See `Lcd_Custom_Config` routine.
Example	`// Write character "e" at current cursor position:` `Lcd_Custom_Chr_Cp('e');`

Lcd_Custom_Cmd

Prototype	`void Lcd_Custom_Cmd(char out_char);`
Description	Sends command to LCD. Parameters : - `out_char`: command to be sent **Note**: Predefined constants can be passed to the function, see Available LCD Commands.
Requires	The LCD module needs to be initialized. See `Lcd_Custom_Config` table.
Example	`// Clear LCD display:` `Lcd_Custom_Cmd(LCD_CLEAR);`

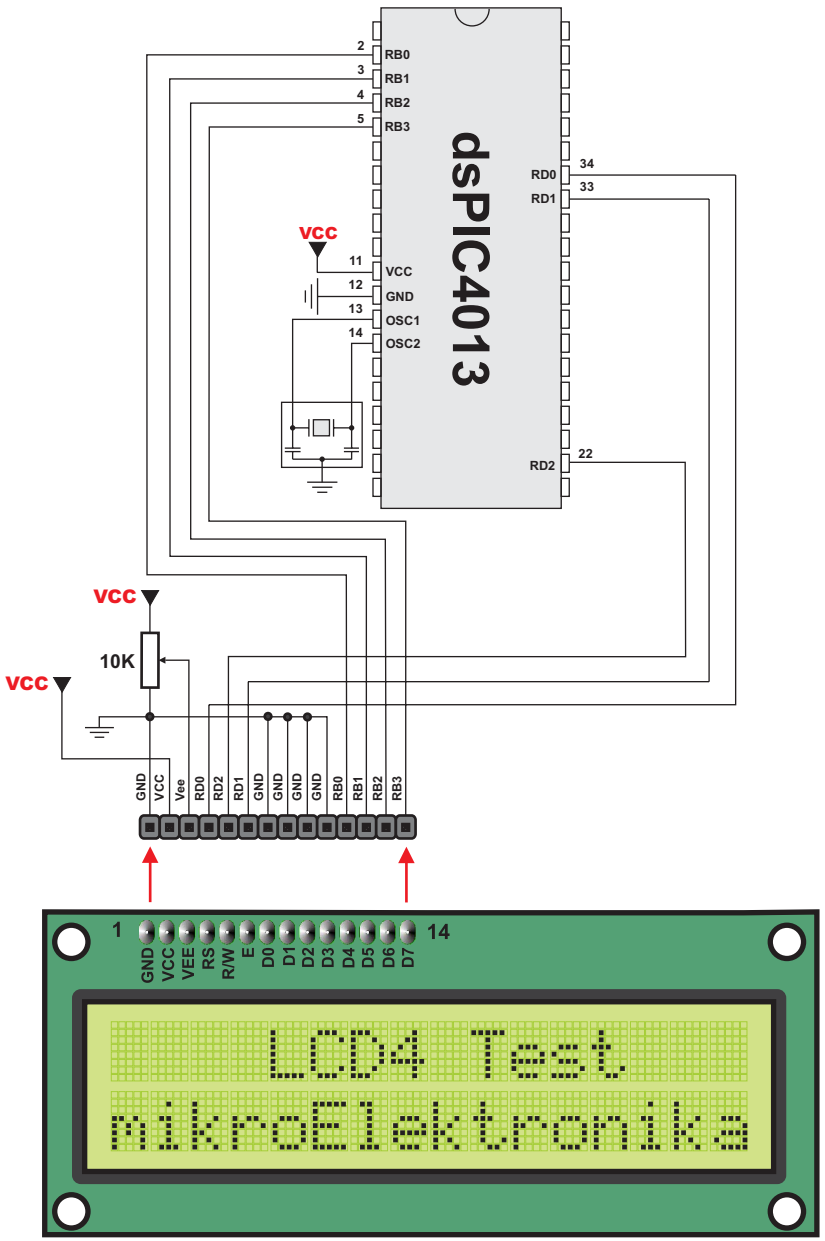
LCD Commands

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

```
char text[ 6]  = "mikro";

void main() {

  //--- PORTB - all digital
  ADPCFG = 0xFFFF;

  Lcd_Custom_Config(&PORTB, 3,2,1,0, &PORTD, 0,2,1);
  Lcd_Custom_Out(1,3, text);
  Lcd_Custom_Out(2,6, text);
  Lcd_Custom_Chr(2,7, 'a');
  Lcd_Custom_Out(1,10, text);
  Lcd_Custom_Chr(1,11, 'o');
} //~!
```

Hardware Connection



LCD 2X16

LCD8 Custom Library (8-bit interface)

The mikroC 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.
Note: mikroElektronika's development system based initialization routines can be found in the setup library files located in the Uses folder.
Note: Only Lcd8_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 Lcd8_Custom_Config call.

Library Routines

```
Lcd8_Custom_Config
Lcd8_Custom_Config_TwoDataPorts
Lcd8_Custom_Out
Lcd8_Custom_Out_Cp
Lcd8_Custom_Chr
Lcd8_Custom_Chr_Cp
Lcd8_Custom_Cmd
```

Lcd8_Custom_Config

Prototype	`void Lcd8_Custom_Config(unsigned int * data_port, unsigned int db7, unsigned int db6, unsigned int db5, unsigned int db4, unsigned int db3, unsigned int db2, unsigned int db1, unsigned int db0, unsigned int * ctrl_port, unsigned int rs, unsigned int ctrl_rw, unsigned int enable);`
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_Custom_Config(&PORTB,7,6,5,4,3,2,1,0, &PORTD,0,1,2);`
	`// or just call the appropriate system based init routine`
	`Lcd8_Custom_Init_EasyDsPIC2();`

Lcd8_Custom_Config_TwoDataPorts

Prototype	`void Lcd8_Custom_Config_TwoDataPorts(unsigned int * data_portHi, unsigned int db7, unsigned int db6, unsigned int db5, unsigned int db4, unsigned int * data_portLo, unsigned int db3, unsigned int db2, unsigned int db1, unsigned int db0, unsigned int * ctrl_port, unsigned int rs, unsigned int ctrl_rw, unsigned int enable);`
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_Custom_Config_TwoDataPorts(&PORTD, 3, 2, 1, 0, &PORTB, 3, 2, 1, 0, &PORTF, 0, 1, 4);`
	
	`// or just call the appropriate system based initialization rou`
	`// tine`
	`Lcd8_Custom_Init_EasyDsPIC3();`

Lcd8_Custom_Out

Prototype	`void Lcd8_Custom_Out(unsigned int row, unsigned int column, char *text);`
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 `Lcd8_Custom_Config` and `Lcd8_Custom_Config_TwoDataPorts` routines.
Example	`// Write text "Hello!" on LCD starting from row 1, column 3:` `Lcd8_Custom_Out(1, 3, "Hello!");`

Lcd8_Custom_Out_CP

Prototype	`void Lcd8_Custom_Out_CP(char *text);`
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 `Lcd8_Custom_Config` and `Lcd8_Custom_Config_TwoDataPorts` routines.
Example	`// Write text "Here!" at current cursor position:` `Lcd8_Custom_Out_CP("Here!");`

Lcd8_Custom_Chr

Prototype	`void Lcd8_Custom_Chr(unsigned int row, unsigned int column, unsigned int out_char);`
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 `Lcd8_Custom_Config` and `Lcd8_Custom_Config_TwoDataPorts` routines.
Example	`// Write character "i" at row 2, column 3:` `Lcd8_Custom_Chr(2, 3, 'i');`

Lcd8_Custom_Chr_CP

Prototype	`void Lcd8_Custom_Chr_CP(unsigned int out_char);`
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 `Lcd8_Custom_Config` and `Lcd8_Custom_Config_TwoDataPorts` routines.
Example	`// Write character "e" at current cursor position:` `Lcd8_Custom_Chr_CP('e');`

Lcd8_Custom_Cmd

Prototype	`void Lcd8_Custom_Cmd(unsigned int out_char);`
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 `Lcd8_Custom_Config` and `Lcd8_Custom_Config_TwoDataPorts` routines.
Example	`// Clear LCD display:` `Lcd8_Custom_Cmd(LCD_CLEAR);`

Library Example

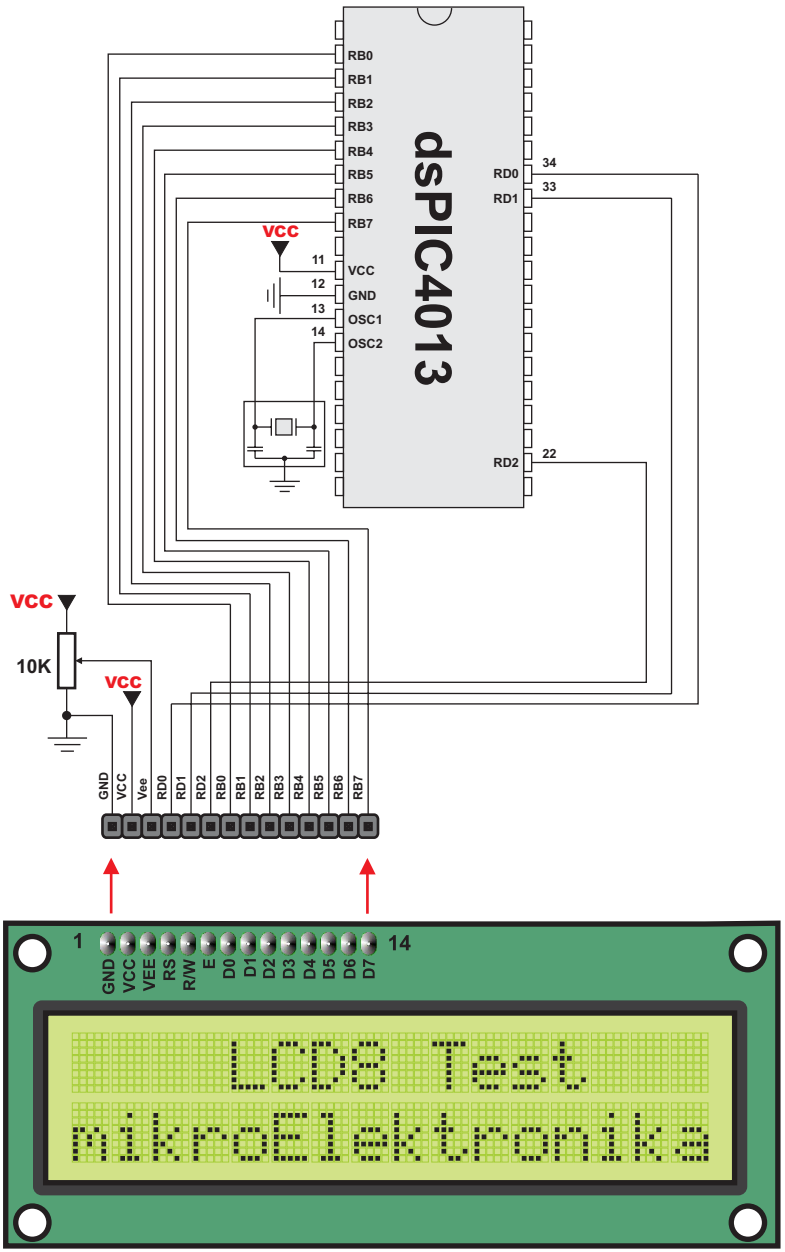
Here is an example of using the Lcd8 Custom Library.

```
void main(){
  ADPCFG = 0xFFFF;  //--- PORTB - all digital

  Lcd8_Custom_Config(&PORTB, 7, 6, 5, 4, 3, 2, 1, 0, &PORTD, 0, 1, 2);
  Lcd8_Custom_Cmd(LCD_CLEAR);
  Lcd8_Custom_Cmd(LCD_CURSOR_OFF);

  Lcd8_Custom_Out(1, 1, "mikroElektronika");
  Lcd8_Custom_Chr(2, 1,'c');
  Lcd8_Custom_Chr_CP('?');
  Lcd8_Custom_Out_CP("for_dsPIC");
} //~!
```

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

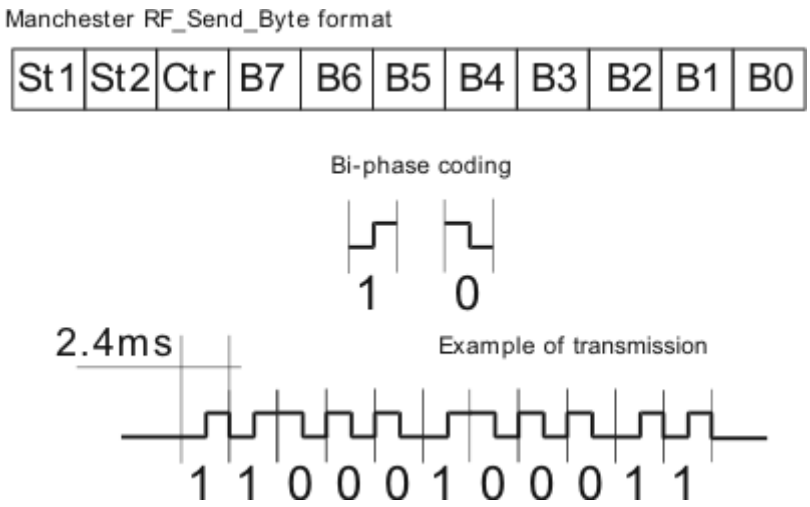
Hardware Connection



LCD 2X16

Manchester Code Library

The mikroC 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).



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 compiler only:

```
Manchester_0
Manchester_1
Manchester_Out
```

Man_Receive_Config

Prototype	`void Man_Receive_Config(unsigned int *port, unsigned int rxpin);`
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	`void Man_Receive_Init(unsigned int *port);`
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_Config(&PORTD);`

Man_Receive

Prototype	`unsigned int Man_Receive(unsigned int *error);`
Returns	One byte from 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	`unsigned int data = 0, error = 0;`
	`...`
	`data = Man_Receive(&error);`
	`if (error)`
	`  { /* error handling */ }`

Man_Send_Config

Prototype	`void Man_Send_Config(unsigned int *port, unsigned int txpin);`
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	`void Man_Send_Init(unsigned int *port);`
Description	The function configures Transmitter pin as pin0 of the MCU's port. Parameters : - `port`: Transmitter port
Requires	Nothing.
Example	`// Configure Transmitter on pin RD0:` `Man_Send_Init(&PORTD);`

Man_Send

Prototype	`void Man_Send(unsigned int data);`
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	`unsigned int msg;`
	`...`
	`Man_Send(msg);`

Man_Synchro

Prototype	`unsigned int Man_Synchro();`
Returns	Half of the manchester bit length, given in multiples of 10us.
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	`unsigned int man_half_bit_len;`
	`...`
	`man_half_bit_len = Man_Synchro();`

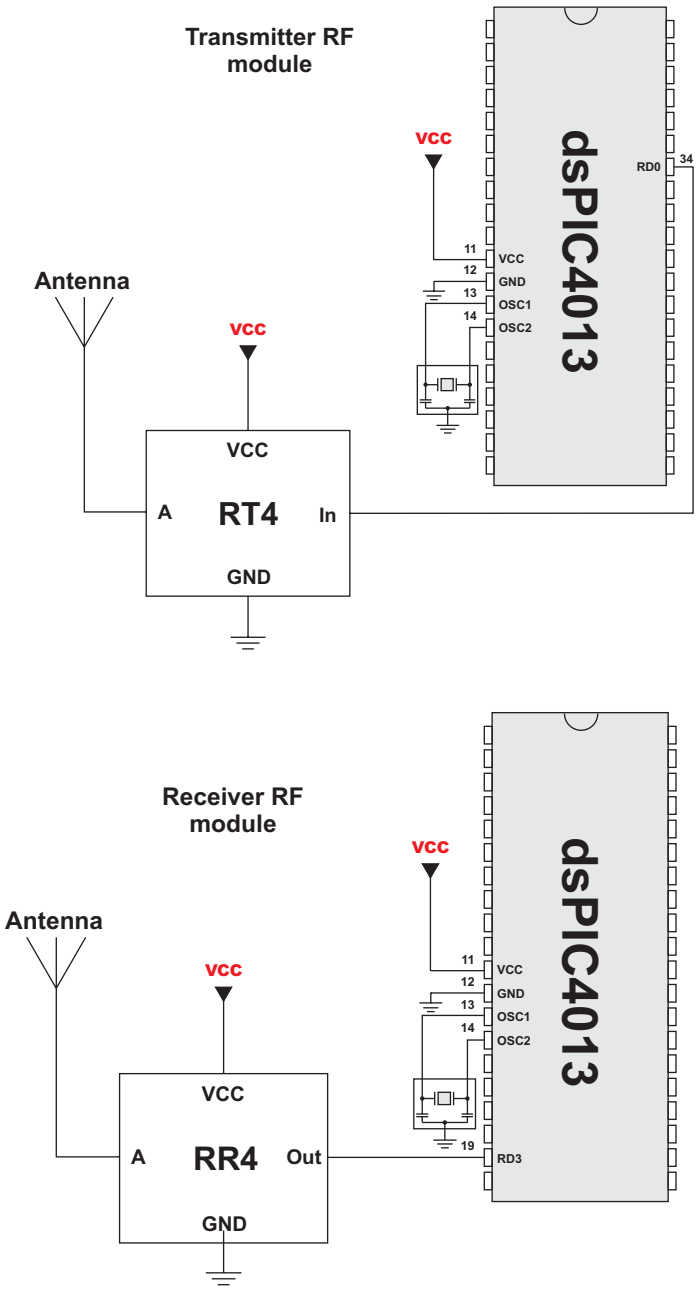
Library Example

The following code is code for the Manchester receiver, it shows how to use the Manchester Library for receiving data:

```
unsigned int
 ERR,
 *error,
 ErrorCount,
 temp;

void main() {
  ADPCFG = 0xFFFF;
  TRISF = 0;
  ERR = 0;
  Uart1_Init(9600);
  error = &ERR;
  ErrorCount = 0;
  Lcd8_Custom_Init_EasyDsPIC3();        // Initialize LCD
  Man_Receive_Config(&PORTD, 3);        // Configure and synchronize receiver
  while (1) {
       Lcd8_Custom_Cmd(LCD_FIRST_ROW);

       while (1)                          // Wait for the start marker
         {
           temp = Man_Receive(error);
           if (temp == 0x0B)
             break;                       // We got the starting sequence
           if (ERR)                       // Exit so we do not loop forever
             break;
         }
       do

//continues ...
```

```
//continued ...

        {
          temp = Man_Receive(error); // Attempt byte receive
          Uart1_Write_Char(0Xff);    // Indicate new reception
          Uart1_Write_Char(temp);    // Display received byte on usart terminal

          if (ERR)
            {
              Lcd8_Custom_Chr_CP('?');
              Uart1_Write_Char('?');
              ErrorCount++;
              if (ErrorCount > 20)
                {
                  //Man_Receive_Init(&PORTD);
                  // alternative:
                  temp = Man_Synchro();
                  ErrorCount = 0;
                }
            }
          else
            {
              if (temp != 0x0E)         // Don't write the } marker on LCD
                Lcd8_Custom_Chr_CP(temp);
            }
          Delay_ms(25);
        }
      while (temp != 0x0E) ;
  }
} //~!
```

Hardware Connection



Multi Media Card Library

The mikroC 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 write operation, make sure you don't 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 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 `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_Read_Sector
Mmc_Write_Sector
Mmc_Read_Cid
Mmc_Read_Csd
```

```
Mmc_Fat_Init
Mmc_Fat_QuickFormat
Mmc_Fat_Assign
Mmc_Fat_Reset
Mmc_Fat_Read
Mmc_Fat_Rewrite
Mmc_Fat_Append
Mmc_Fat_Delete
Mmc_Fat_Write
Mmc_Fat_Set_File_Date
Mmc_Fat_Get_File_Date
Mmc_Fat_Get_File_Size
Mmc_Fat_Get_Swap_File
```

Mmc_Init

Prototype	`unsigned Mmc_Init(unsigned *port, unsigned cspin);`
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.

Example	`// Initialize the SPI module`
	`Spi_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);`
	`// Loop until MMC is initialized`
	`while (Mmc_Init(&PORTG, 9))`
	`  ;`
	
	`// Reinitialize the SPI module at higher speed (change primary prescaler).`
	`Spi_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_Read_Sector

Prototype	`unsigned Mmc_Read_Sector(unsigned long sector, char *dbuff);`
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. - `dbuff`: 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`
	`unsigned int error;`
	`unsigned long sectorNo = 510;`
	`char dataBuffer[512];`
	`...`
	`error = Mmc_Read_Sector(sectorNo, dataBuffer);`

Mmc_Write_Sector

Prototype	`unsigned Mmc_Write_Sector(unsigned long sector, char *dbuff);`
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. - `dbuff`: 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`
	`unsigned int error;`
	`unsigned long sectorNo = 510;`
	`char dataBuffer[512];`
	`...`
	`error = Mmc_Write_Sector(sectorNo, dataBuffer);`

Mmc_Read_Cid

Prototype	`unsigned Mmc_Read_Cid(char *data_cid);`
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	`unsigned int error;`
	`char dataBuffer[16];`
	`...`
	`error = Mmc_Read_Cid(dataBuffer);`

Mmc_Read_Csd

Prototype	`unsigned Mmc_Read_Csd(char *data_csd);`
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	`unsigned int error;`
	`char dataBuffer[16];`
	`...`
	`error = Mmc_Read_Csd(dataBuffer);`

Mmc_Fat_Init

Prototype	`unsigned Mmc_Fat_Init(unsigned *port, unsigned pin);`
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`
	`Spi_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`
	`Mmc_Fat_Init(&PORTG, 9);`
	
	`// Reinitialize the SPI module at higher speed (change primary prescaler).`
	`Spi_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_QuickFormat

Prototype	`unsigned Mmc_Fat_QuickFormat(unsigned *port, unsigned pin, char * mmc_fat_label);`
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. **Note**: This routine can be used instead or in conjunction with `Mmc_Fat_Init` routine. **Note**: If MMC/SD card already contains a valid boot sector, it will remain unchanged (except volume label field) and only FAT and ROOT tables will be erased. Also, the new volume label will be set.
Requires	The appropriate hardware SPI module must be previously initialized.

Example	`// Initialize the SPI module`
	`Spi_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(&PORTG, 9, "mikroE");`
	`// Reinitialize the SPI module at higher speed`
	`// (change primary prescaler).`
	`Spi_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	`unsigned Mmc_Fat_Assign(char *filename, char file_cre_attr);`
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 no have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with 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	`void Mmc_Fat_Reset(unsigned long *size);`
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	`unsigned long size;`
	`...`
	`Mmc_Fat_Reset(size);`

Mmc_Fat_Read

Prototype	`void Mmc_Fat_Read(unsigned char *bdata);`
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	`char character;`
	`...`
	`Mmc_Fat_Read(&character);`

Mmc_Fat_Rewrite

Prototype	`void 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	`void 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 write 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	`void 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	`void Mmc_Fat_Write(char *fdata, unsigned data_len);`
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	`char file_contents[ 42] ;`
	`...`
	`Mmc_Fat_Write(file_contents, 42);`
	`// write data to the assigned file`

Mmc_Set_File_Date

Prototype	`void Mmc_Fat_Set_File_Date(unsigned int year, unsigned short month, unsigned short day, unsigned short hours, unsigned short mins, unsigned short seconds);`
Description	Sets the date/time stamp. Any subsequent file write 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	`void Mmc_Fat_Get_File_Date(unsigned int *year, unsigned short *month, unsigned short *day, unsigned short *hours, unsigned short *mins);`
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	`unsigned year;`
	`char month, day, hours, mins;`
	`...`
	`Mmc_Fat_Get_File_Date(&year, &month, &day, &hours, &mins);`

Mmc_Fat_Get_File_Size

Prototype	`unsigned long Mmc_Fat_Get_File_Size();`
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	`unsigned long my_file_size;`
	`...`
	`my_file_size = Mmc_Fat_Get_File_Size();`

Mmc_Fat_Get_Swap_File

Prototype	`unsigned long Mmc_Fat_Get_Swap_File(unsigned long sectors_cnt, char* filename, char file_attr);`
Returns	- Number of the start sector for the newly created swap file, if there was enough free space on the MMC/SD card to create file of required size. - 0 - otherwise.
Description	This function is used to create a swap file of predefined name and size on the MMC/SD media. If a file with specified name already exists on the media, search for consecutive sectors will ignore sectors occupied by this file. Therefore, it is 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. It's main purpose in the mikroC's library is to be used for fast data acquisition; when the time-critical acquisition has finished, the data can be re-written into a "normal" file, and formatted in the most suitable way. Parameters: - `sectors_cnt`: number of consecutive sectors that user wants the swap file to have. - `filename`: name of the file that should be assigned for file operations. File name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does no have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with 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_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
unsigned long size;
...
size = Mmc_Fat_Get_Swap_File(1000, "mikroE.txt", 0x20);
if (size) {
  Usart_Write(0xAA);
  Usart_Write(Lo(size));
  Usart_Write(Hi(size));
  Usart_Write(Higher(size));
  Usart_Write(Highest(size));
  Usart_Write(0xAA);
} //~
```

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.

```
#include <spi_const.h>

const char SWAP_FILE_MSG[ ]  = "Swap file at: ";

char
 fat_txt[ 20]  = "FAT16 not found",
 file_contents[ 50]  = "XX MMC/SD FAT16 library by Anton Rieckertn";

char
 filename[ 14]  = "MIKRO00xTXT";           // File names
unsigned
 loop, loop2;
unsigned short
 caracter;
unsigned long
 i, size;
char Buffer[ 512] ;

//I-I-I--------- Writes string to USART
void I_Write_Str(char *ostr) {
  unsigned i;

  i = 0;
  while (ostr[ i] ) {
    Uart1_Write_Char(ostr[ i++] );
  }
} //~

//M-M-M--------- Creates a new file and writes some data to it
void 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(loop = 1; loop <= 99; loop++) {    //  We want 5 files on the MMC card
    Uart1_Write_Char('.');
    file_contents[ 0]  = loop / 10 + 48;
    file_contents[ 1]  = loop % 10 + 48;
    Mmc_Fat_Write(file_contents, 42);    // write data to the assigned file
  }
} //~

//continues...
```

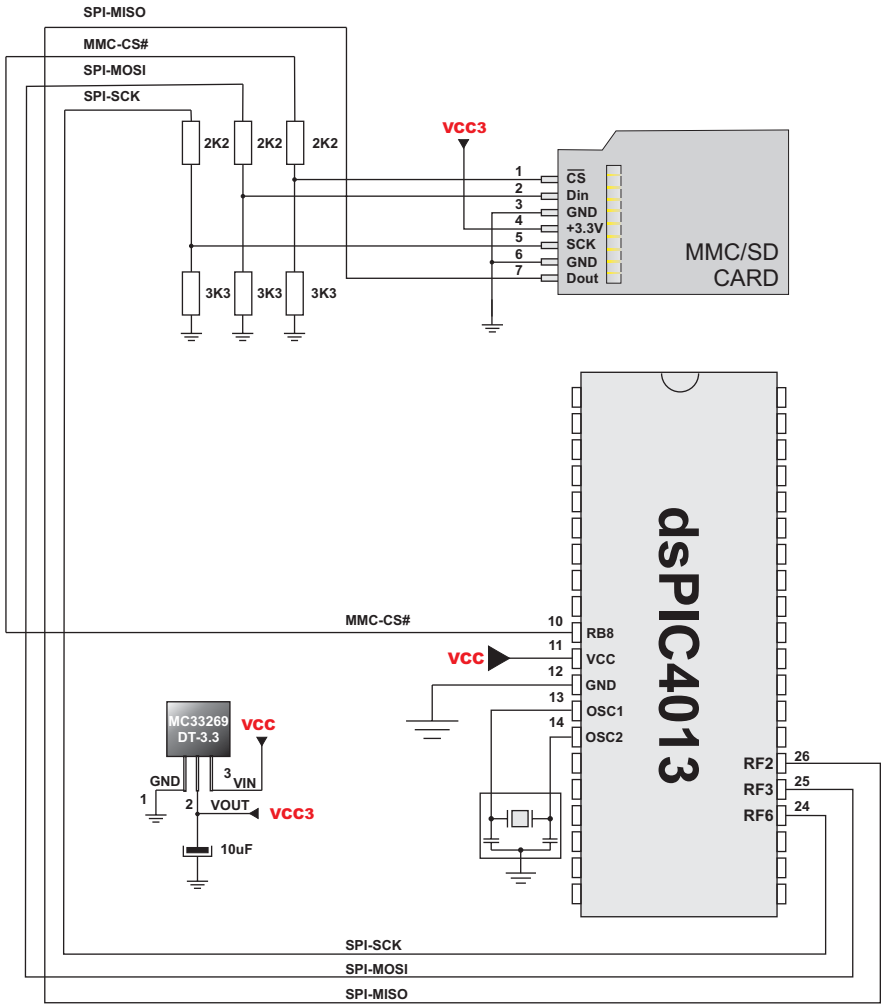
```
// continued...

//M-M-M--------- Creates many new files and writes data to them
void M_Create_Multiple_Files() {
  for(loop2 = 'B'; loop2 <= 'Z'; loop2++) {
    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(loop = 1; loop <= 44; loop++) {
      file_contents[ 0]  = loop / 10 + 48;
      file_contents[ 1]  = loop % 10 + 48;
      Mmc_Fat_Write(file_contents, 42);  // write data to the assigned file
    }
  }
} //~

//M-M-M--------- Opens an existing file and rewrites it
void M_Open_File_Rewrite() {
  filename[ 7]  = 'C';
  Mmc_Fat_Assign(&filename, 0);
  Mmc_Fat_Rewrite();
  for(loop = 1; loop <= 55; loop++) {
    file_contents[ 0]  = loop / 10 + 64;
    file_contents[ 1]  = loop % 10 + 64;
    Mmc_Fat_Write(file_contents, 42);     // write data to the assigned file
  }
} //~

//M-M-M--------- Opens an existing file and appends data to it
//                (and alters the date/time stamp)
void 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 2005n", 27);
     // Write data to the assigned file
} //~

//continues...
```

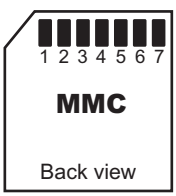
```
// continued...
//M-M-M--------- Opens an existing file, reads data from it and puts it to USART
void M_Open_File_Read() {
  filename[7] = 'B';
  Mmc_Fat_Assign(&filename, 0);
  Mmc_Fat_Reset(&size);         // To read file, procedure returns size of file
  for (i = 1; i <= size; i++) {
    Mmc_Fat_Read(&caracter);
    Uart1_Write_Char(caracter);// Write data to USART
  }
}//~

//M-M-M--------- Deletes a file. If the file doesn't exist,
//it will first be created
//and then deleted.
void M_Delete_File() {
  filename[7] = 'F';
  Mmc_Fat_Assign(filename, 0);
  Mmc_Fat_Delete();
}//~

//M-M-M--------- Tests whether file exists, and if so sends its creation date
//                and file size via USART
void M_Test_File_Exist(char fLetter) {
  unsigned long fsize;
  unsigned int year;
  unsigned short month, day, hour, minute;
  unsigned char outstr[12];

  filename[7] = fLetter;
  if (Mmc_Fat_Assign(filename, 0)) {
    //--- file has been found - get its date
    Mmc_Fat_Get_File_Date(&year, &month, &day, &hour, &minute);
    WordToStr(year, outstr);
    I_Write_Str(outstr);
    ByteToStr(month, outstr);
    I_Write_Str(outstr);
    WordToStr(day, outstr);
    I_Write_Str(outstr);
    WordToStr(hour, outstr);
    I_Write_Str(outstr);
    WordToStr(minute, outstr);
    I_Write_Str(outstr);
    //--- get file size
    fsize = Mmc_Fat_Get_File_Size();
    LongToStr((signed long)fsize, outstr);
    I_Write_Str(outstr);
  }
 //continues...
```

```
// continued...
  else {
    //--- file was not found - signal it
    Uart1_Write_Char(0x55);
    Delay_ms(1000);
    Uart1_Write_Char(0x55);
  }
}//~

//--------------- Tries to create a swap file, whose size will be at least 100
//                 sectors (see Help for details)
void M_Create_Swap_File() {
  unsigned int i;

  for(i=0; i<512; i++)
    Buffer[ i]  = i;

  size = Mmc_Fat_Get_Swap_File(5000, "mikroE.txt", 0x20);
  // see help on this function for details

  if (size) {
    LongToStr((signed long)size, fat_txt);
    I_Write_Str(fat_txt);

    for(i=0; i<5000; i++) {
      Mmc_Write_Sector(size++, Buffer);
      Uart1_Write_Char('.');
    }
  }
}//~
// Main. Uncomment the function(s) to test the desired
// operation(s)
void main() {
     //--- prepare PORTD for signalling
     PORTD = 0;
     TRISD = 0;
     ADPCFG = 0xFFFF;
     //--- set up USART for the file read
     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);

     Uart_Init(19200);
     U1MODEbits.ALTIO = 1;       // clear the way for SPI
     Delay_ms(200);             // wait for the UART module to stabilize

//continues...
```

```
// continued...
      //--- init the FAT library
      if (!Mmc_Fat_Init(&PORTB,8)) {
           // 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);
           //--- Test start
           PORTD = 0x0005;

           // --- Test routines. Uncomment
           // them one-by-one to test certain features
           M_Create_New_File();
           M_Create_Multiple_Files();

           M_Open_File_Rewrite();
           M_Open_File_Append();
           M_Delete_File();
           M_Create_Swap_File();

           M_Open_File_Read();
           M_Test_File_Exist('F');          // this file will not exist here
           M_Test_File_Exist('B');          // this file will exist here

    }
      else {
        I_Write_Str(fat_txt);
      }
      //--- Test termination
      PORTD = 0xFFFF;
} //~!
```

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	`unsigned int Ow_Reset(unsigned int *port, unsigned int pin);`
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	`unsigned short Ow_Read(unsigned int *port, unsigned int pin);`
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`
	`// unsigned short read_data;`
	`...`
	`read_data = Ow_Read(&PORTF, 6);`

Ow_Write

Prototype	`void Ow_Write(unsigned int *port, unsigned int pin, unsigned short par);`
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);`

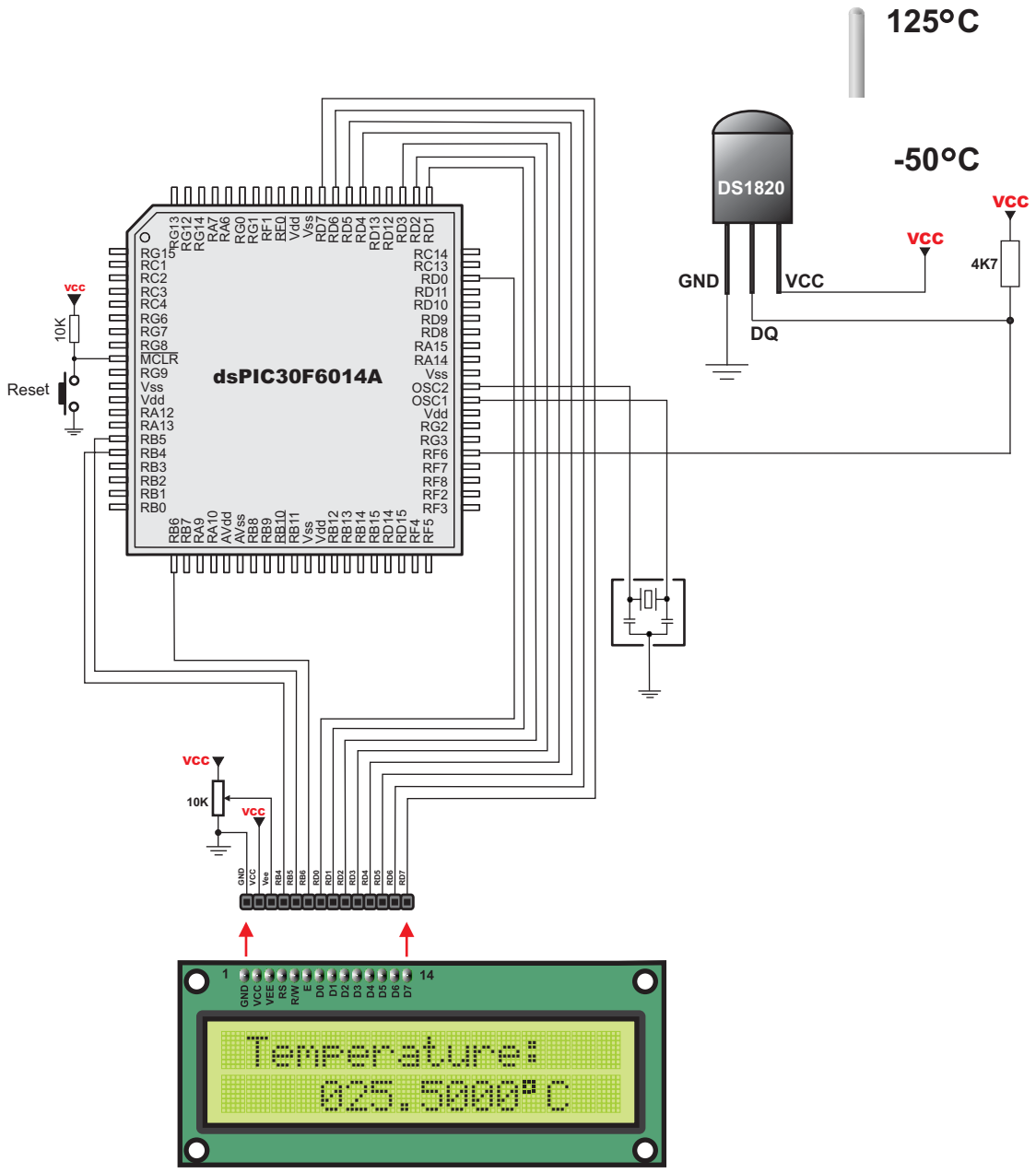
Library Example

This example reads the temperature using DS18x20 connected to pin RF6. After reset, MCU obtains temperature from the sensor and prints it on the LCD. Be sure to set Fosc appropriately in your project, to pull-up RF6 line and to turn off the PORTF leds.

```
// Set TEMP_RESOLUTION to the corresponding resolution of your DS18x20 sensor:
//  18S20: 9  (default setting; can be 9,10,11,or 12)
//  18B20: 12
const unsigned short TEMP_RESOLUTION = 9;

char *text = "000.0000";
unsigned temp;

void Display_Temperature(unsigned int temp2write) {
  const unsigned short RES_SHIFT = TEMP_RESOLUTION - 8;
  char temp_whole;
  unsigned int temp_fraction;

  // check if temperature is negative
  if (temp2write & 0x8000) {
     text[ 0] = '-';
     temp2write = ~temp2write + 1;
     }

  // extract temp_whole
  temp_whole = temp2write >> RES_SHIFT ;

  // convert temp_whole to characters
  if (temp_whole/100)
     text[ 0] = temp_whole/100  + 48;

  text[ 1] = (temp_whole/10)%10 + 48;
  text[ 2] =  temp_whole%10     + 48;

  // extract temp_fraction
  temp_fraction = temp2write << (4-RES_SHIFT);
  temp_fraction &= 0x000F;
  temp_fraction *= 625;

  // convert temp_fraction to characters
  text[ 4] =  temp_fraction/1000    + 48;
  text[ 5] = (temp_fraction/100)%10 + 48;
  text[ 6] = (temp_fraction/10)%10  + 48;
  text[ 7] =  temp_fraction%10      + 48;


// continues ...
```

```
// continued ...

  // print temperature on LCD
  Lcd8_Custom_Out(2, 5, text);
}//~

void main() {

  ADPCFG = 0xFFFF;

  Lcd8_Custom_Config(&PORTD,7,6,5,4,3,2,1,0, &PORTB, 4, 5, 6);
  Lcd8_Custom_Cmd(LCD_CURSOR_OFF);
  Lcd8_Custom_Out(1, 1, " Temperature:    ");
  // Print degree character, 'C' for Centigrades
  Lcd8_Custom_Chr(2,13,223);
  Lcd8_Custom_Chr(2,14,'C');

  //--- main loop
  do {
    //--- perform temperature reading
    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

    temp =  Ow_Read(&PORTF,6);
    temp = (Ow_Read(&PORTF,6) << 8) + temp;

    //--- Format and display result on Lcd
    Display_Temperature(temp);

    Delay_ms(500);
  } while (1);
}//~!
```

Hardware Connection



LCD 2X16

Port Expander Library

The mikroC 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	`void Expander_Init(char ModuleAddress, unsigned int *rstport, unsigned int rstpin, unsigned int *csport, unsigned int cspin);`
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 `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	`char Expander_Read_Byte(char ModuleAddress, char RegAddress);`
Returns	Byte read.
Description	The function reads byte from Port Expander. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `RegAddress`: Port Expander's internal register address.
Requires	Port Expander must be initialized. See `Expander_Init`.

Example	`// Read a byte from Port Expander's register`
	`char read_data;`
	`...`
	`read_data = Expander_Read_Byte(0,1);`

Expander_Write_Byte

Prototype	`void Expander_Write_Byte(char ModuleAddress, char RegAddress, char Data);`
Returns	Nothing.
Description	Routine writes a byte to Port Expander. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `RegAddress`: Port Expander's internal register address - `Data`: data to be written
Requires	Port Expander must be initialized. See `Expander_Init`.
Example	`// Write a byte to the Port Expander's register` `Expander_Write_Byte(0,1,$FF);`

Expander_Read_PortA

Prototype	`char Expander_Read_PortA(char ModuleAddress);`
Returns	Byte read.
Description	The function reads byte from Port Expander's PortA. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page.
Requires	Port Expander must be initialized. See `Expander_Init`. Port Expander's PortA should be configured as input. See `Expander_Set_DirectionPortA` and `Expander_Set_DirectionPortAB` routines.

Example	`// Read a byte from Port Expander's PORTA`
	`char read_data;`
	`...`
	`Expander_Set_DirectionPortA(0,0xFF);`
	`// set expander's porta to be input`
	`...`
	`read_data = Expander_Read_PortA(0);`

Expander_Read_PortB

Prototype	`char Expander_Read_PortB(char ModuleAddress);`
Returns	Byte read.
Description	The function reads byte from Port Expander's PortB. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page
Requires	Port Expander must be initialized. See `Expander_Init`. Port Expander's PortB should be configured as input. See `Expander_Set_DirectionPortB` and `Expander_Set_DirectionPortAB` routines.

Example	`// Read a byte from Port Expander's PORTB`
	`char read_data;`
	`...`
	`Expander_Set_DirectionPortB(0,0xFF);`
	`// set expander's portb to be input`
	`...`
	`read_data = Expander_Read_PortB(0);`

Expander_Read_PortAB

Prototype	`unsigned int Expander_Read_PortAB(char ModuleAddress);`
Returns	Word read.
Description	The function reads word from Port Expander's ports. PortA readings are in the higher byte of the result. PortB readings are in the lower byte of the result. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page.
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA and PortB should be configured as inputs. See `Expander_Set_DirectionPortA`, `Expander_Set_DirectionPortB` and `Expander_Set_DirectionPortAB` routines.

Example	`// Read a byte from Port Expander's PORTA and PORTB`
	`unsigned int read_data;`
	`...`
	`Expander_Set_DirectionPortAB(0,0xFFFF);`
	`// set expander's porta and portb to be input`
	`...`
	`read_data = Expander_Read_PortAB(0);`

Expander_Write_PortA

Prototype	`void Expander_Write_PortA(char ModuleAddress, char Data);`
Returns	Nothing.
Description	The function writes byte to Port Expander's PortA. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written
Requires	Port Expander must be initialized. See `Expander_Init`. Port Expander's PortA should be configured as output. See `Expander_Set_DirectionPortA` and `Expander_Set_DirectionPortAB` routines.

Example	`// Write a byte to Port Expander's PORTA`
	`...`
	`Expander_Set_DirectionPortA(0,0x00);`
	`// set expander's porta to be output`
	`...`
	`Expander_Write_PortA(0, 0xAA);`

Expander_Write_PortB

Prototype	`void Expander_Write_PortB(char ModuleAddress, char Data);`
Returns	Nothing.
Description	The function writes byte to Port Expander's PortB. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data to be written
Requires	Port Expander must be initialized. See `Expander_Init`. Port Expander's PortB should be configured as output. See `Expander_Set_DirectionPortB` and `Expander_Set_DirectionPortAB` routines.

Example	`// Write a byte to Port Expander's PORTB`
	
	`...`
	`Expander_Set_DirectionPortB(0,0x00);`
	`// set expander's portb to be output`
	`...`
	`Expander_Write_PortB(0, 0x55);`

Expander_Write_PortAB

Prototype	`void Expander_Write_PortAB(char ModuleAddress, unsigned int Data);`
Returns	Nothing.
Description	The function writes word to Port Expander's ports. Data to be written to PortA 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 - `Data`: data to be written. Data to be written to PortA are passed in Data's higher byte. Data to be written to PortB are passed in `Data`'s lower byte
Requires	Port Expander must be initialized. See `Expander_Init`. Port Expander's PortA and PortB should be configured as outputs. See `Expander_Set_DirectionPortA`, `Expander_Set_DirectionPortB` and `Expander_Set_DirectionPortAB` routines.

Example	`// Write a byte to Port Expander's PORTA and PORTB`
	
	`...`
	`Expander_Set_DirectionPortAB(0,0x0000);`
	`// set expander's porta and portb to be output`
	`...`
	`Expander_Write_PortAB(0, 0xAA55);`

Expander_Set_DirectionPortA

Prototype	`void Expander_Set_DirectionPortA(char ModuleAddress, char Data);`
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 designates corresponding pin as input. Cleared bit designates 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(0,0x00);`

Expander_Set_DirectionPortB

Prototype	`void Expander_Set_DirectionPortB(char ModuleAddress, char Data);`
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 designates corresponding pin as input. Cleared bit designates 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(0,0xFF);`

Expander_Set_DirectionPortAB

Prototype	`void Expander_Set_DirectionPortAB(char ModuleAddress, unsigned int Direction);`
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 designates corresponding pin as input. Cleared bit designates 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(0,0x00FF);`

Expander_Set_PullUpsPortA

Prototype	`void Expander_Set_PullUpsPortA(char ModuleAddress, char Data);`
Description	The function sets Port Expander's PortA pull up/down resistors. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data for choosing pull up/down resistors configuration. Each bit corresponds to the appropriate pin of the PortA register. Set bit enables pull-up for corresponding pin.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`// Set Port Expander's PORTA pull-up resistors` `Expander_Set_PullUpsPortA(0, 0xFF);`

Expander_Set_PullUpsPortB

Prototype	`void Expander_Set_PullUpsPortB(char ModuleAddress, char Data);`
Description	The function sets Port Expander's PortB pull up/down resistors. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data`: data for choosing pull up/down resistors configuration. Each bit corresponds to the appropriate pin of the PortB register. Set bit enables pull-up for corresponding pin.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`// Set Port Expander's PORTB pull-up resistors` `Expander_Set_PullUpsPortB(0, 0xFF);`

Expander_Set_PullUpsPortAB

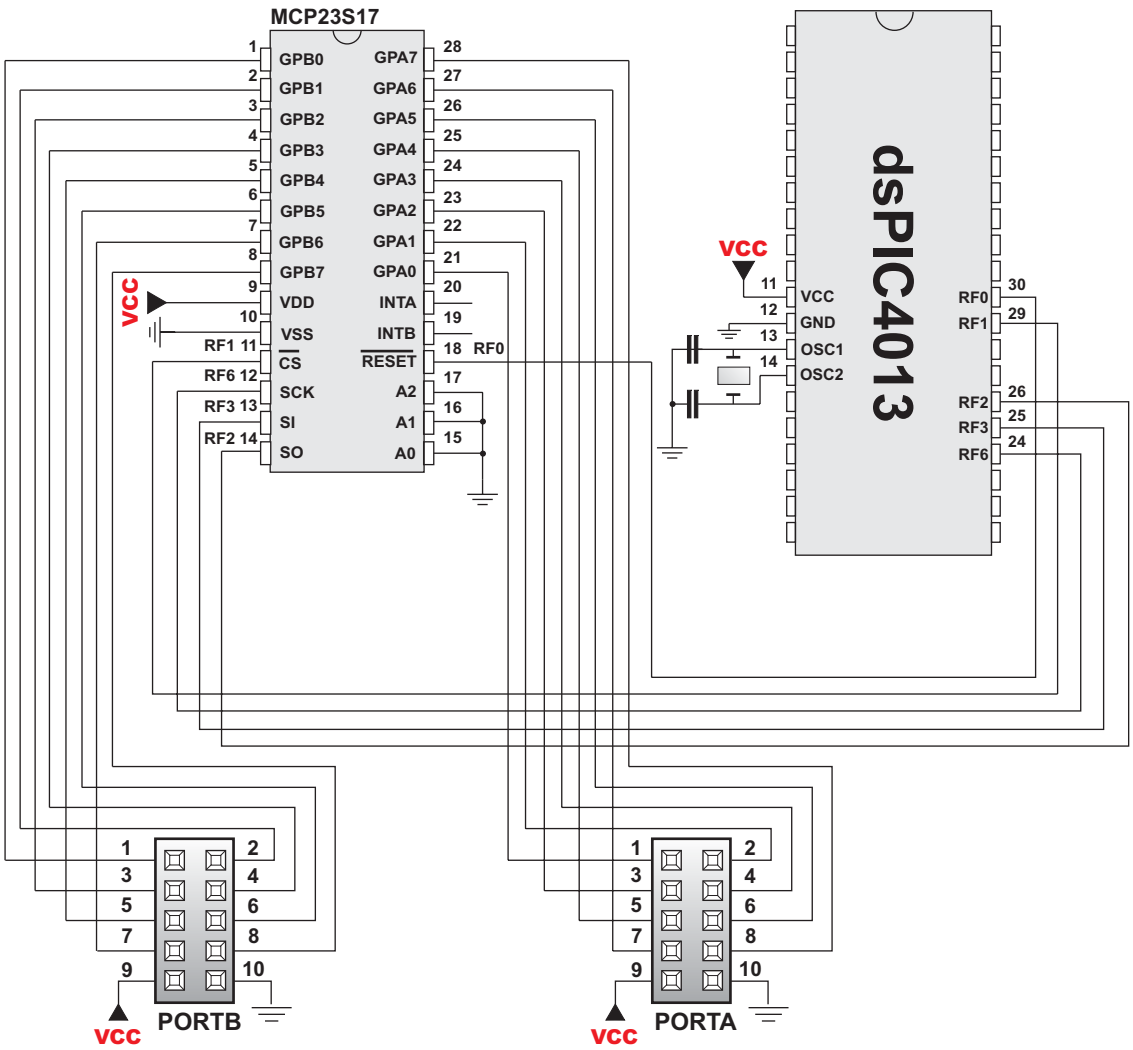
Prototype	`void Expander_Set_PullUpsPortAB(char ModuleAddress, unsigned int PullUps);`
Description	The function sets Port Expander's PortA and PortB pull up/down resistors. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `PullUps`: data for choosing pull up/down resistors configuration. PortA pull up/down resistors configuration is passed in PullUps's higher byte. PortB pull up/down resistors configuration is passed in PullUps's lower byte. Each bit corresponds to the appropriate pin of the PortA/PortB register. Set bit enables pull-up for corresponding pin.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`// Set Port Expander's PORTA and PORTB pull-up resistors` `Expander_Set_PullUpsPortAB(0, 0xFFFF);`

Library Example

The example demonstrates how to communicate with Port Expander MCP23S17. Note that Port Expander pins A2 A1 A0 are connected to GND so Port Expander Hardware Address is 0.

```
unsigned int  i;

void main(){
  ADPCFG = 0xFFFF;
  TRISB = 0x00;
  LATB  = 0xFF;
  Delay_ms(2000);

  Spi1_Init();                                  // initialize SPI1 module
  Expander_Init(0, &PORTF, 0, &PORTF, 1);   // initialize port expander
  Expander_Set_DirectionPortA(0, 0);         // set expander's porta to be output
  Expander_Set_DirectionPortB(0,0xFF);     // set expander's portb to be input
  Expander_Set_PullUpsPortB(0,0xFF); // set pull ups to the expander's portb pins
  i = 0;
  while(1) {
       Expander_Write_PortA(0, i++); // write i to expander's porta
       LATB = Expander_Read_PortB(0);
       // read expander's portb and write it to the LATB register
       if(i == 255)
         i = 0;
       Delay_ms(20);

  }
} //~
```

Hardware Connection



PS/2 Library

The mikroC 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	`void Ps2_Init(unsigned int *port);`
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	`void Ps2_Config(unsigned int *port, unsigned int clkpin, unsigned int datapin);`
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	`unsigned int Ps2_Key_Read(unsigned int *value, unsigned int *special, unsigned int *pressed);`
Returns	- `1` if reading of a key from the keyboard was successful - `0` if no key was pressed
Description	The function retrieves information on key pressed. Parameters : - `value`: holds the value of the key pressed. For characters, numerals, punctuation marks, and space value will store the appropriate ASCII code. Routine “recognizes” the function of Shift and Caps Lock, and behaves appropriately. For special function keys see Special Function Keys Table. - `special`: is a flag for special function keys (F1, Enter, Esc, etc). If key pressed is one of these, special will be set to 1, otherwise 0. - `pressed`: is set to 1 if the key is pressed, and 0 if it is released.
Requires	PS/2 keyboard needs to be initialized. See `Ps2_Init` and `Ps2_Config` routines.

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

Special Function Keys

Key	Value Returned
F1	1
F2	2
F3	3
F4	4
F5	5
F6	6
F7	7
F8	8
F9	9
F10	10
F11	11
F12	12
Enter	13
Page Up	14
Page Down	15
Backspace	16

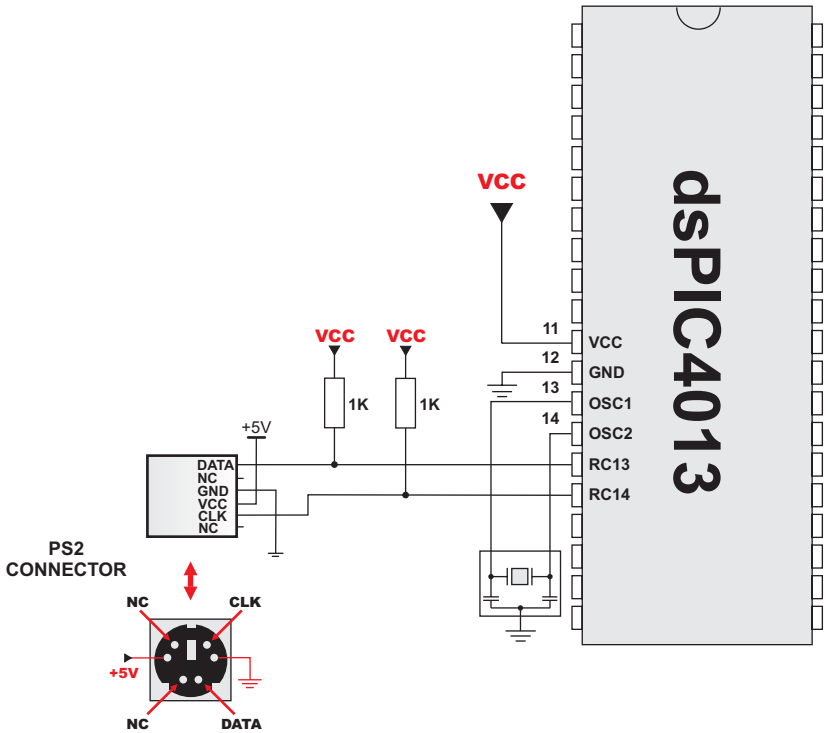
Key	Value Returned
Insert	17
Delete	18
Windows	19
Ctrl	20
Shift	21
Alt	22
Print Screen	23
Pause	24
Caps Lock	25
End	26
Home	27
Scroll Lock	28
Num Lock	29
Left Arrow	30
Right Arrow	31
Up Arrow	32
Down Arrow	33
Escape	34
Tab	35

Library Example

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

```
unsigned int
 keydata = 0, special = 0, down = 0;

void main() {
  ADPCFG = 0xFFFF;
  Uart1_Init(9600);
  Ps2_Init(&PORTC);          // Init PS/2 Keyboard on PORTC
                             // pin 13 is connected to Data line
                             // pin 14 is connected to Clock line

  Delay_ms(100);             // Wait for keyboard to finish
  Uart1_Write_Char('s'); Uart1_Write_Char('t'); Uart1_Write_Char('a');
  Uart1_Write_Char('r'); Uart1_Write_Char('t'); Uart1_Write_Char('!');
  do {
    if(Ps2_Key_Read(&keydata, &special, &down)) {
      if(down && (keydata == 16)) { // Backspace
        Uart1_Write_Char(0x08);
      }
      else if(down && (keydata == 13)) { // Enter
        Uart1_Write_Char('r');       // send carriage return to usart terminal
        //Uart1_Write_Char('n');
        // uncomment this line if usart terminal also expects line feed
                                           // for new line transition
      }
      else if(down && !special && keydata) {
        //Uart1_Write_Char(keydata >> 8);
        Uart1_Write_Char(keydata);
      }
    }
    Delay_ms(1);// debounce
  } while(1);
} //~
```

Hardware Connection



PWM Library

The CCP module is available with a number of dsPIC30/33 and PIC24 MCUs. The mikroC 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	`unsigned int Pwm_Init( unsigned int freq_hz , unsigned int enable_channel_x, unsigned int timer_prescale, unsigned int use_timer_x);`
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.
Requires	MCU must have the HW PWM Module.

Example	`// Initializes the PWM module at 5KHz, channel 1, no clock`
	`// prescale, timer2 :`
	`unsigned int pwm_period1;`
	`...`
	`pwm_period1 = Pwm_Init(5000, 1, 0, 2);`

Pwm_Set_Duty

Prototype	`void Pwm_Set_Duty(unsigned duty, unsigned channel);`
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 `Pwm_Init` routine.

Example	`// Set channel 1 duty ratio to 50%:`
	`unsigned int pwm_period1;`
	`...`
	`Pwm_Set_Duty(pwm_period1/2, 1);`

Pwm_Start

Prototype	`void Pwm_Start(char enable_channel_x);`
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

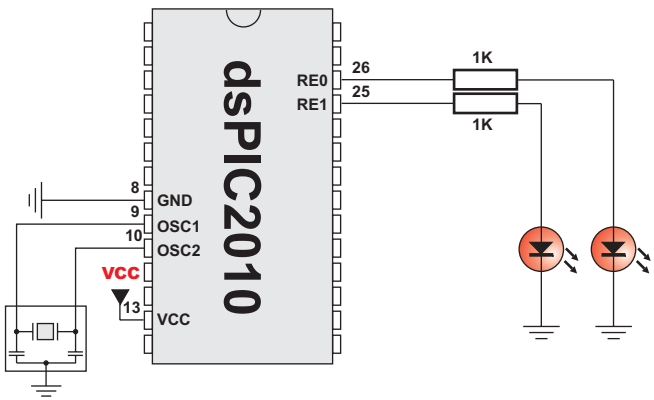
Prototype	`void Pwm_Stop(char disable_channel_x);`
Description	Stops PWM at requested channel. Parameters : - `disable_channel_x`: number of PWM channel **Note**: number of available PWM channels depends on MCU. Refer to MCU datasheet for details.
Requires	MCU must have the HW PWM Module.
Example	`// stop PWM at channel 1` `Pwm_Stop(1);`

Library Example

The example changes PWM duty ratio on channels 1 and 2 continually. If LEDs are connected to channels 1 and 2, gradual change of emitted light will be noticeable.

```
void main() {
  unsigned pwm_period1, pwm_period2, i1 = 0, i2 = 0;

  pwm_period1 = Pwm_Init(5000, 1, 0, 2);
  pwm_period2 = Pwm_Init(10000, 2, 0, 3);

  Pwm_Start(1);
  Pwm_Start(2);

  while(1) {
    Pwm_Set_Duty(i1, 1);
    Pwm_Set_Duty(i2, 2);
    if (i1++ == pwm_period1)
      i1 = 0;
    if (i2++ == pwm_period2)
      i2 = 0;
    Delay_ms(1);
  }
} //~
```

Hardware Connection



PWM Motor Library

The PWM Motor Control module is available with a number of dsPIC30/33 MCUs. The mikroC 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	`unsigned int Pwm_Mc_Init(unsigned int freq_hz , unsigned int pair_output_mode, unsigned int enable_output_x, unsigned int clock_prescale_output_postscale);`
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:*/`
	`unsigned int duty_50;`
	`...`
	`Pwm_Mc_Init(5000, 1, 0x0F, 0);`

Pwm_Mc_Set_Duty

Prototype	`void Pwm_Mc_Set_Duty(unsigned duty, unsigned channel);`
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(5000,1,0xF,0);`
	`Pwm_Mc_Set_Duty(32767, 1);`

Pwm_Mc_Start

Prototype	`void Pwm_Mc_Start(void);`
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

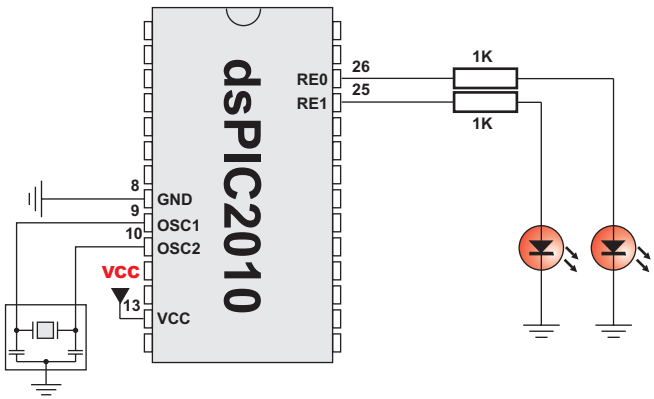
Prototype	`void Pwm_Mc_Stop(void);`
Description	Stops the Motor Control PWM module.
Requires	The dsPIC30/33 MCU must have the Motor Control PWM module.
Example	`// stop the Motor Control PWM module` `Pwm_Mc_Stop();`

Library Example

The example changes PWM duty ratio on channel 1 continually. If LED is connected to the channel 1, a gradual change of emitted light will be noticeable.

```
unsigned int i;
unsigned int duty_50;

void main(){
       ADPCFG = 0xFFFF;
       PORTB = 0xAAAA;
       TRISB = 0;
       Delay_ms(1000);

       duty_50 = Pwm_Mc_Init(5000,1,0x01,0);
       // Pwm_Mc_Init returns 50% of the duty
       Pwm_Mc_Set_Duty(i = duty_50,1);
       Pwm_Mc_Start();

       do
       {
          i--;
          Pwm_Mc_Set_Duty(i,1);
          Delay_ms(1);
          if (i == 0)
            i = duty_50 * 2 - 1;
            // Let us not allow  overflow
          PORTB = i;
       }
       while(1);

} //~
```

Hardware Connection



RS-485 Library

RS-485 is a multipoint communication which allows multiple devices to be connected to a single bus. The mikroC 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	`void Rs485master_Init(unsigned int *port, unsigned int pin);`
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 `Uart_Init`.
Example	`// intialize mcu as a Master for RS-485 communication` `Rs485master_Init(&PORTF, 4);`

RS485Master_Receive

Prototype	`void RS485Master_Receive(unsigned short *data);`
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	`char msg[ 8];`
	`...`
	`RS485Master_Receive(msg);`

RS485Master_Send

Prototype	`void RS485Master_Send(unsigned short *data, unsigned short datalen, unsigned short address);`
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	`char msg[8];`
	`...`
	`// send 3 bytes of data to slave with address 0x12`
	`RS485Master_Send(msg, 3, 0x12);`

RS485Slave_Init

Prototype	`void Rs485Slave_Init(unsigned int *port, unsigned int pin, char address);`
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	`// intialize mcu as a Slave for RS-485 with address 160` `Rs485slave_Init(&PORTF, 4, 160);`

RS485Slave_Receive

Prototype	`void RS485Slave_Receive(unsigned short *data);`
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	`char msg[ 8];`
	`...`
	`RS485Slave_Read(msg);`

RS485Slave_Send

Prototype	`void RS485Slave_Send(unsigned short *data, unsigned short datalen);`
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	`char msg[ 8];`
	`...`
	`// send 2 bytes of data to the master`
	`RS485Slave_Send(msg, 2);`

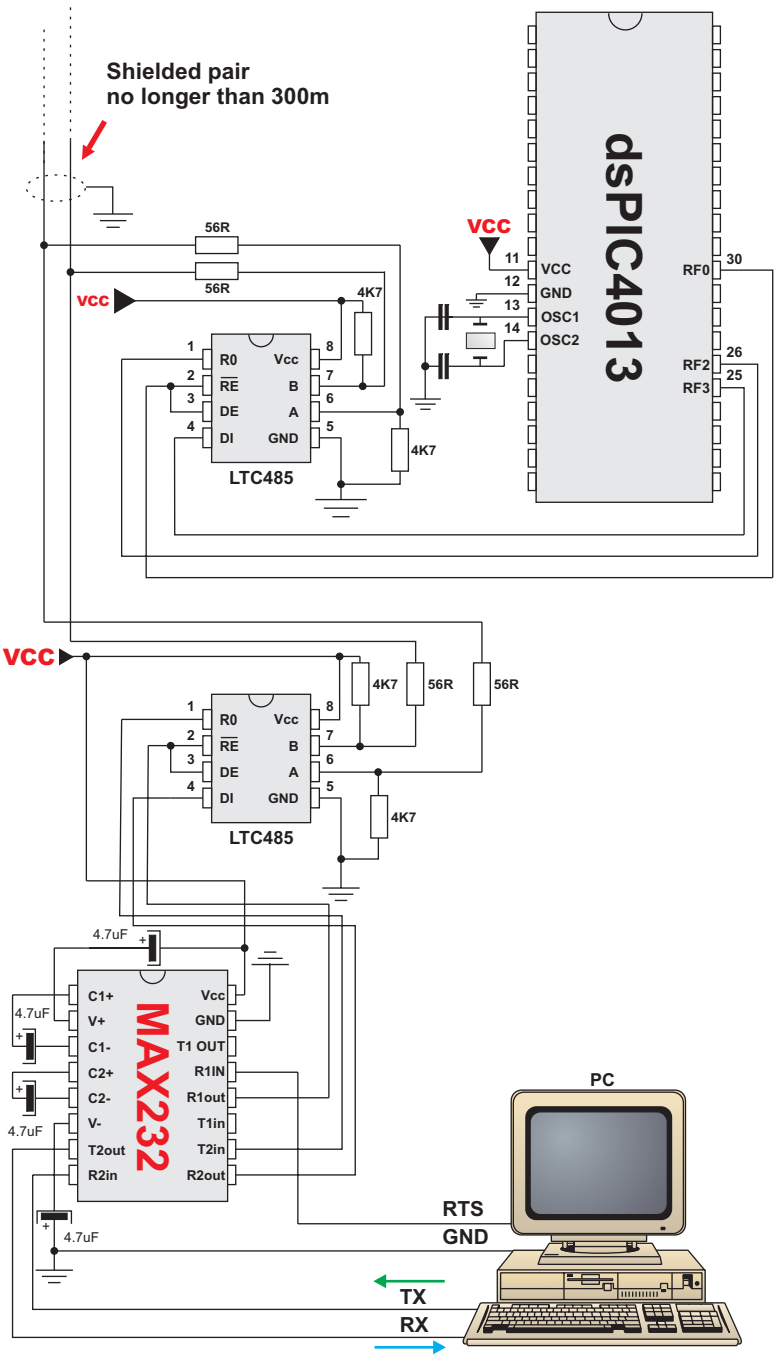
Library Example

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.

```
char dat[10];               // buffer for receving/sending messages
int i,j;
long you = 0;

//-------------- Interrupt routine
void interrupt_uart() org 0x26 {
  Rs485master_Receive(dat);
  IFS0bits.U1RXIF = 0;                    //ensure interrupt not pending
} //~!~

void main() {
  ADPCFG = 0xFFFF;
  PORTB = 0;
  TRISB = 0;
  PORTD = 0;
  TRISD = 0;

  Uart1_Init(9600);
  Rs485master_Init(&PORTF, 4);           // intialize mcu as a 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;
  INTCON1bits.NSTDIS = 1;               //no nesting of interrupts
  IFS0bits.U1RXIF = 0;                  //ensure interrupt not pending
  IEC0bits.U1RXIE = 1;                  //enable intterupt

// continues ...
```

```
// continued ...

  while (1){
        // upon completed valid message receiving
        //   data[4] is set to 255
    you++;
    if (dat[5])  {     // if error detected, signal it by
      PORTD = 0xAA;    //   setting PORTD to 0xAA
    }
    if (dat[4]) {
                       // if message received successfully
      you = 0;
      dat[4]  = 0;     //   clear message received flag
      j = dat[3];
      for (i = 1; i <= j; i++) {
        PORTB = dat[i-1]; // show data on PORTB
      }                   // increment received dat[0]
      dat[0]  = dat[0]+1;   //   and send it to slave
      Delay_ms(10);
      Rs485master_Send(dat,1,160);
    }
    if (you > 100000) {   // if in 100000 poll-cycles the answer

      PORTD++;           //   was not detected, signal

      you = 0;           //   failure of send-message
      Rs485master_Send(dat,1,160);
      if (PORTD > 10) {   // if sending failed 10 times
        PORTD = 0;
        Rs485master_Send(dat,1,50);
        // send message on broadcast address
      }
    }
  }
} //~!
```

Hardware Connection



Software I2C Library

The mikroC 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	`void Soft_I2C_Init(unsigned int *portOut, int sda, int clk);`
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	`void Soft_I2C_Start(void);`
Description	Determines if the I²C bus is free and issues START signal.
Requires	Software I²C must be configured before using this function. See `Soft_I2C_Init` routine.
Example	`// Issue START signal` `Soft_I2C_Start();`

Soft_I2C_Read

Prototype	`unsigned short Soft_I2C_Read(unsigned int ack);`
Returns	One byte from the Slave.
Description	Reads one byte from the slave. Parameters : - `ack`: acknowledge signal parameter. If the `ack==0` not acknowledge signal will be sent after reading, otherwise the acknowledge signal will be sent.
Requires	Soft I²C must be configured before using this function. See `Soft_I2C_Init` routine. Also, START signal needs to be issued in order to use this function. See `Soft_I2C_Start` routine.

Example	`unsigned short take;`
	`...`
	`// Read data and send the not_acknowledge signal`
	`take = Soft_I2C_Read(0);`

Soft_I2C_Write

Prototype	`unsigned short Soft_I2C_Write(unsigned short Data);`
Returns	- `0` if there were no errors. - `1` if write collision was detected on the 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 `Soft_I2C_Init` routine. Also, START signal needs to be issued in order to use this function. See `Soft_I2C_Start` routine.

Example	`unsigned short data, error;`
	`...`
	`error = Soft_I2C_Write(data);`
	`error = Soft_I2C_Write(0xA3);`

Soft_I2C_Stop

Prototype	`void Soft_I2C_Stop(void);`
Description	Issues STOP signal.
Requires	Soft I²C must be configured before using this function. See 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 sends date/time to RTC.

```
void main() {

  ADPCFG = 0xFFFF;

  Soft_I2C_Init(&PORTB, 4, 3);
// Initialize Software I2C communication (full master mode) on pins RB4 and RB3
  Soft_I2C_Start();          // Issue start signal
  Soft_I2C_Write(0xA0);      // Address PCF8583
  Soft_I2C_Write(0);         // Start from word at address 0 (configuration word)
  Soft_I2C_Write(0x80);      // Write 0x80 to config. (pause counter...)
  Soft_I2C_Write(0);         // Write 0 to cents word
  Soft_I2C_Write(0);         // Write 0 to seconds word
  Soft_I2C_Write(0x30);      // Write 0x30 to minutes word
  Soft_I2C_Write(0x11);      // Write 0x11 to hours word
  Soft_I2C_Write(0x30);      // Write 0x24 to year/date word
  Soft_I2C_Write(0x08);      // Write 0x08 to weekday/month
  Soft_I2C_Stop();           // Issue stop signal

  Soft_I2C_Start();          // Issue start signal
  Soft_I2C_Write(0xA0);      // Address PCF8583
  Soft_I2C_Write(0);         // Start from word at address 0
  Soft_I2C_Write(0);         // Write 0 to config word (enable counting)
  Soft_I2C_Stop();           // Issue stop signal

} //~!
```

Software SPI Library

The mikroC 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	`void Soft_Spi_Config(unsigned int *port, const unsigned short SDI, const unsigned short SD0, const unsigned short SCK);`
Description	Configures and initializes the software SPI module. Parameters : - `portOut`: software SPI port address - `SDI`: serial **data input** line pin - `SD0`: 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_Read

Prototype	`void Soft_Spi_Init(unsigned int *port);`
Returns	Nothing.
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.
Requires	Nothing.

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	`unsigned short Soft_Spi_Read(char data);`
Returns	Nothing.
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 `Soft_Spi_Config` and `Soft_Spi_Init` routines.

Example	`unsigned short data_read;`
	`char data_send;`
	`...`
	`// Read a byte and assign it to data_read variable`
	`// (data_send byte will be sent via SPI during the Read opera`
	`// tion)`
	`data_read = Soft_Spi_Read(data_send);`

Soft_Spi_Write

Prototype	`void Soft_Spi_Write(char data);`
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 `Soft_Spi_Config` and `Soft_Spi_Init` routines.
Example	`// Write a byte to the Soft SPI bus` `Soft_Spi_Write(0xAA);`

Library Example

This code demonstrates using library routines for Soft_SPI communication. Also, this example demonstrates working with max7219. Eight 7 segment displays are connected to MAX7219. MAX7219 is connected to SDO, SDI, SCKL pins are connected accordingly.

```
//MAX 7219 chip_util Library
//Defines-------------
#define CHIP_SELECT F1
//---------end-defines

void max7219_init1() {
  PORTB.CHIP_SELECT = 0;          // SELECT MAX
  Soft_SPI_Write(0x09);           // BCD mode for digit decoding
  Soft_Spi_Write(0xFF);
  PORTB.CHIP_SELECT = 1;          // DESELECT MAX

  PORTB.CHIP_SELECT = 0;          // SELECT MAX
  Soft_Spi_Write(0x0A);
  Soft_Spi_Write(0x0F);           // Segment luminosity intensity
  PORTB.CHIP_SELECT = 1;          // DESELECT MAX

  PORTB.CHIP_SELECT = 0;          // SELECT MAX
  Soft_Spi_Write(0x0B);
  Soft_Spi_Write(0x07);           // Display refresh
  PORTB.CHIP_SELECT = 1;          // DESELECT MAX

// continues ...
```

```
// continues ...

  PORTB.CHIP_SELECT = 0;                    // SELECT MAX
  Soft_Spi_Write(0x0C);
  Soft_Spi_Write(0x01);                     // Turn on the display
  PORTB.CHIP_SELECT = 1;                    // DESELECT MAX

  PORTB.CHIP_SELECT = 0;                    // SELECT MAX
  Soft_Spi_Write(0x00);
  Soft_Spi_Write(0xFF);                     // No test
  PORTB.CHIP_SELECT = 1;                    // DESELECT MAX
}

unsigned int i;

void main() {
  ADPCFG = 0xFFFF;
  TRISB.CHIP_SELECT = 0;
  Delay_ms(100);
  Soft_Spi_Init(&PORTB);                    // standard configuration
  //  Soft_Spi_Config(&PORTB, 4,5,3);
  // configure individual pins
  max7219_init1();

  for (i = 1; i<=8u; i++) {
    PORTB.CHIP_SELECT = 0;   // select max7219
    Soft_Spi_Write(i);       // send i to max7219 (digit place)
    Soft_Spi_Write(8-i);     // send i to max7219 (digit)
    PORTB.CHIP_SELECT = 1;   // deselect max7219
  }

} //~!
```

Software UART Library

The mikroC 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	`unsigned Soft_Uart_Init(unsigned int *port, unsigned int rx, unsigned int tx, unsigned long baud_rate, unsigned int inverted);`
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	`char Soft_Uart_Read(int *error);`
Returns	Byte received via UART.
Description	The function receives a byte via software UART. This is a blocking function call (waits for start bit). 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	`char data;`
	`int error;`
	`...`
	`// wait until data is received`
	`do`
	`  data = Soft_Uart_Read(&error);`
	`while (error);`
	
	`// Now we can work with data:`
	`if (data) { ...}`

Soft_Uart_Write

Prototype	`void Soft_Uart_Write(char data);`
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	`char some_byte = 0x0A;`
	`...`
	`// Write a byte via Soft Uart`
	`Soft_Uart_Write(some_byte);`

Library Example

This example demonstrates simple data exchange via software UART. If MCU is connected to the PC, you can test the example from the mikroC for dsPIC30/33 and PIC24 USART communication terminal, launch it from the drop-down menu Tools › USART Terminal or simply click the USART Terminal Icon.

```
void main() {
      char *txt = "mikroe";
      int i = 0;
      int error;

      // Initialize Software Uart communication
      Soft_Uart_Init(&PORTF, 2,3, 56000, 0);

      // Write text to Software Uart
      while(txt[ i] ){
        Soft_Uart_Write(txt[ i++] );
      }

      // Receive byte and send it back via Software Uart
      while (1){
        i = Soft_Uart_Read(&error);
        if (!error)
          Soft_Uart_Write(i);
        else
          Soft_Uart_Write('e'),Soft_Uart_Write(i);
      }
} //~!
```

Sound Library

The mikroC 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	`void Sound_Init(unsigned *snd_port, unsigned snd_pin);`
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	`void Sound_Play(unsigned freq_in_hz, unsigned duration_ms);`
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)
Requires	In order to hear the sound, you need a piezo speaker (or other hardware) on designated port. Also, you must call Sound_Init to prepare hardware for output before using this function.
Example	`// Play sound of 1KHz in duration of 100ms` `Sound_Play(1000, 100);`

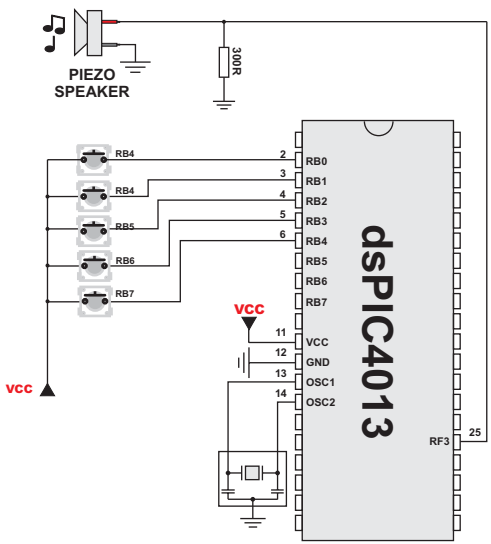
Library Example

The example is a simple demonstration of how to use the Sound Library for playing tones on a piezo speaker.

```
void Tone1() {
  Sound_Play(500, 200); // Period = 2ms <=> 500Hz, Duration = 200ms
}//~
void Tone2() {
  Sound_Play(555, 200); // Period = 1.8ms <=> 555Hz
}//~
void Tone3() {
  Sound_Play(625, 200); // Period = 1.6ms <=> 625Hz
}//~
void Melody() {          // Plays the melody "Yellow house"
  Tone1(); Tone2(); Tone3(); Tone3();
  Tone1(); Tone2(); Tone3(); Tone3();
  Tone1(); Tone2(); Tone3();
  Tone1(); Tone2(); Tone3(); Tone3();
  Tone1(); Tone2(); Tone3();
  Tone3(); Tone3(); Tone2(); Tone2(); Tone1();
}//~
void ToneA() {
  Sound_Play(1250, 20);
}
void ToneC() {
  Sound_Play(1450, 20);
}
void ToneE() {
  Sound_Play(1650, 80);
}

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

void main() {
  ADPCFG = 0xFFFF;      // initialize AN pins as digital
  TRISB = 0x1F;

  Sound_Init(&PORTF, 3);
  Sound_Play(2000, 100);

// continues ...
```

```
// continued ...

  while (1) {                     // endless loop
    if (Button(&PORTB,4,1,1))     // RB4 plays Tone1
      Tone1();
    while (PORTB & 0x10) ;        // Wait for button to be released

    if (Button(&PORTB,3,1,1))     // RB3 plays Tone2
      Tone2();
    while (PORTB & 0x08) ;        // Wait for button to be released

    if (Button(&PORTB,2,1,1))     // RB2 plays Tone3
      Tone3();
    while (PORTB & 0x04) ;

    if (Button(&PORTB,1,1,1))     // RB1 plays Melody2
      Melody2();
    while (PORTB & 0x02) ;         // Wait for button to be released

    if (Button(&PORTB,0,1,1))     // RB0 plays Melody
      Melody();
    while (PORTB & 0x01) ;        // Wait for button to be released
  }
} //~!
```

Hardware Connection



SPI Library

The SPI module is available with all dsPIC30/33 and PIC24 MCUs. The mikroC 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            Spi2_Init            Spi_Init
Spi1_Init_Advanced   Spi2_Init_Advanced   Spi_Init_Advanced
Spi1_Read            Spi2_Read            Spi_Read
Spi1_Write           Spi2_Write           Spi_Write
                                          Spi_Set_Active
```

Spi1_Init

Prototype	`void Spi1_Init(void);`
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	`void Spi1_Init_Advanced(unsigned master_mode, unsigned mode16, unsigned sec_prescaler, unsigned pri_prescaler, unsigned slave_select, unsigned data_sample, unsigned clock_idle, unsigned edge);`

Description	Configures and initializes the SPI1 module.
	The parameter `master_mode` determines the working mode for SPI1. It can have the following values:
	
	`//--- Master/Slave mode, SPI1CON1<5>`
	`_SPI_MASTER = 0x0020, // Master mode`
	`_SPI_SLAVE = 0x0000, // 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>`
	`_SPI_16_BIT = 0x0400, // 16-bit mode`
	`_SPI_8_BIT = 0x0000, // 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>`
	`_SPI_PRESCALE_SEC_1 = 0x001C, // secondary prescale 1:1`
	`_SPI_PRESCALE_SEC_2 = 0x0018, // secondary prescale 2:1`
	`_SPI_PRESCALE_SEC_3 = 0x0014, // secondary prescale 3:1`
	`_SPI_PRESCALE_SEC_4 = 0x0010, // secondary prescale 4:1`
	`_SPI_PRESCALE_SEC_5 = 0x000C, // secondary prescale 5:1`
	`_SPI_PRESCALE_SEC_6 = 0x0008, // secondary prescale 6:1`
	`_SPI_PRESCALE_SEC_7 = 0x0004, // secondary prescale 7:1`
	`_SPI_PRESCALE_SEC_8 = 0x0000, // secondary prescale 8:1`
	
	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>`
	`_SPI_PRESCALE_PRI_1 = 0x0003, // primary prescale 1:1`
	`_SPI_PRESCALE_PRI_4 = 0x0002, // primary prescale 4:1`
	`_SPI_PRESCALE_PRI_16 = 0x0001, // primary prescale 16:1`
	`_SPI_PRESCALE_PRI_64 = 0x0000, // primary prescale 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.
	`//continues on the next page ...`

<table>
<tr><td></td><td>

```
//continued from the previous page ...
```

It can be:

```
    //--- Slave select enable bit (Slave Mode), SPI1CON1<7>
    _SPI_SS_ENABLE = 0x0080,  // SS used for the Slave mode
    _SPI_SS_DISABLE = 0x0000,  // 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>
    _SPI_DATA_SAMPLE_MIDDLE = 0x0000,
    // data sampled in the middle of data output time
    _SPI_DATA_SAMPLE_END = 0x2000,
    // 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>
    _SPI_CLK_IDLE_LOW  = 0x0000,
    // IDLE state is Lo, ACTIVE state is Hi
    _SPI_CLK_IDLE_HIGH = 0x0040,
    // IDLE state is Hi, ACTIVE state is Lo
```

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>
    _SPI_ACTIVE_2_IDLE         = 0x0000,
    // data is valid on ACTIVE-to-IDLE transition
    // (data changes on IDLE-to-ACTIVE transition)
    _SPI_IDLE_2_ACTIVE         = 0x0100;
    // data is valid on IDLE-to-ACTIVE transition
    // (data changes on ACTIVE-to-IDLE transition)
```

</td></tr>
<tr><td>Requires</td><td>MCU must have the SPI1 module.</td></tr>
<tr><td>Example</td><td>

```
/* 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);
```

</td></tr>
</table>

Spi1_Read

Prototype	`unsigned Spi1_Read(unsigned data_out);`
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`
	`char take, buffer;`
	`...`
	`take = Spi1_Read(buffer);`

Spi1_Write

Prototype	`void Spi1_Write(unsigned data_out);`
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`
	`char buffer;`
	`...`
	`Spi1_Write(buffer);`

Spi2_Init

Prototype	`void Spi2_Init(void);`
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	`void Spi2_Init_Advanced(unsigned master_mode, unsigned mode16, unsigned sec_prescaler, unsigned pri_prescaler, unsigned slave_select, unsigned data_sample, unsigned clock_idle, unsigned edge);`

Description	Configures and initializes the SPI2 module.
	The parameter master_mode determines the working mode for SPI2. It can have the following values:
	
	`//--- Master/Slave mode, SPI2CON1<5>`
	`_SPI_MASTER = 0x0020, // Master mode`
	`_SPI_SLAVE = 0x0000, // 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>`
	`_SPI_16_BIT = 0x0400, // 16-bit mode`
	`_SPI_8_BIT = 0x0000, // 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>`
	`_SPI_PRESCALE_SEC_1 = 0x001C, // secondary prescale 1:1`
	`_SPI_PRESCALE_SEC_2 = 0x0018, // secondary prescale 2:1`
	`_SPI_PRESCALE_SEC_3 = 0x0014, // secondary prescale 3:1`
	`_SPI_PRESCALE_SEC_4 = 0x0010, // secondary prescale 4:1`
	`_SPI_PRESCALE_SEC_5 = 0x000C, // secondary prescale 5:1`
	`_SPI_PRESCALE_SEC_6 = 0x0008, // secondary prescale 6:1`
	`_SPI_PRESCALE_SEC_7 = 0x0004, // secondary prescale 7:1`
	`_SPI_PRESCALE_SEC_8 = 0x0000, // secondary prescale 8:1`
	
	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>`
	`_SPI_PRESCALE_PRI_1 = 0x0003, // primary prescale 1:1`
	`_SPI_PRESCALE_PRI_4 = 0x0002, // primary prescale 4:1`
	`_SPI_PRESCALE_PRI_16 = 0x0001, // primary prescale 16:1`
	`_SPI_PRESCALE_PRI_64 = 0x0000, // primary prescale 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.
	
	`//continues on the next page ...`

<table>
<tr><td></td><td>

```
//continued from the previous page ...
```

It can be:

```
    //--- Slave select enable bit (Slave Mode), SPI2CON1<7>
    _SPI_SS_ENABLE  = 0x0080,  // SS used for the Slave mode
    _SPI_SS_DISABLE = 0x0000,  // 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>
    _SPI_DATA_SAMPLE_MIDDLE    = 0x0000,
    // data sampled in the middle of data output time
    _SPI_DATA_SAMPLE_END       = 0x2000,
    // 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>
    _SPI_CLK_IDLE_LOW  = 0x0000,
    // IDLE state is Lo, ACTIVE state is Hi
    _SPI_CLK_IDLE_HIGH = 0x0040,
    // IDLE state is Hi, ACTIVE state is Lo
```

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>
    _SPI_ACTIVE_2_IDLE         = 0x0000,
    // data is valid on ACTIVE-to-IDLE transition
    // (data changes on IDLE-to-ACTIVE transition)
    _SPI_IDLE_2_ACTIVE         = 0x0100;
    // data is valid on IDLE-to-ACTIVE transition
    // (data changes on ACTIVE-to-IDLE transition)
```

</td></tr>
<tr><td>Requires</td><td>MCU must have the SPI2 module.</td></tr>
<tr><td>Example</td><td>

```
/*Set SPI2 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:*/
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);
```

</td></tr>
</table>

Spi2_Read

Prototype	`unsigned Spi2_Read(unsigned data_out);`
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 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`
	`char take, buffer;`
	`...`
	`take = Spi2_Read(buffer);`

Spi2_Write

Prototype	`void Spi2_Write(unsigned data_out);`
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`
	`char buffer;`
	`...`
	`Spi2_Write(buffer);`

Spi_Init

Prototype	`void Spi_Init(void);`
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	`void Spi_Init_Advanced(unsigned master_mode, unsigned mode16, unsigned sec_prescaler, unsigned pri_prescaler, unsigned slave_select, unsigned data_sample, unsigned clock_idle, unsigned edge);`

Description	Configures and initializes the SPI1 module.
	The parameter `master_mode` determines the working mode for SPI1. It can have the following values:
	
	`//--- Master/Slave mode, SPI1CON1<5>`
	`_SPI_MASTER = 0x0020, // Master mode`
	`_SPI_SLAVE = 0x0000, // 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>`
	`_SPI_16_BIT = 0x0400, // 16-bit mode`
	`_SPI_8_BIT = 0x0000, // 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>`
	`_SPI_PRESCALE_SEC_1 = 0x001C, // secondary prescale 1:1`
	`_SPI_PRESCALE_SEC_2 = 0x0018, // secondary prescale 2:1`
	`_SPI_PRESCALE_SEC_3 = 0x0014, // secondary prescale 3:1`
	`_SPI_PRESCALE_SEC_4 = 0x0010, // secondary prescale 4:1`
	`_SPI_PRESCALE_SEC_5 = 0x000C, // secondary prescale 5:1`
	`_SPI_PRESCALE_SEC_6 = 0x0008, // secondary prescale 6:1`
	`_SPI_PRESCALE_SEC_7 = 0x0004, // secondary prescale 7:1`
	`_SPI_PRESCALE_SEC_8 = 0x0000, // secondary prescale 8:1`
	
	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>`
	`_SPI_PRESCALE_PRI_1 = 0x0003, // primary prescale 1:1`
	`_SPI_PRESCALE_PRI_4 = 0x0002, // primary prescale 4:1`
	`_SPI_PRESCALE_PRI_16 = 0x0001, // primary prescale 16:1`
	`_SPI_PRESCALE_PRI_64 = 0x0000, // primary prescale 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.
	//continues on the next page ...

<table>
<tr><td></td><td>

```
//continued from the previous page ...
```

It can be:

```
    //--- Slave select enable bit (Slave Mode), SPI1CON1<7>
    _SPI_SS_ENABLE = 0x0080,  // SS used for the Slave mode
    _SPI_SS_DISABLE = 0x0000,  // 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>
    _SPI_DATA_SAMPLE_MIDDLE = 0x0000,
    // data sampled in the middle of data output time
    _SPI_DATA_SAMPLE_END = 0x2000,
    // 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>
    _SPI_CLK_IDLE_LOW  = 0x0000,
    // IDLE state is Lo, ACTIVE state is Hi
    _SPI_CLK_IDLE_HIGH = 0x0040,
    // IDLE state is Hi, ACTIVE state is Lo
```

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>
    _SPI_ACTIVE_2_IDLE         = 0x0000,
    // data is valid on ACTIVE-to-IDLE transition
    // (data changes on IDLE-to-ACTIVE transition)
    _SPI_IDLE_2_ACTIVE         = 0x0100;
    // data is valid on IDLE-to-ACTIVE transition
    // (data changes on ACTIVE-to-IDLE transition)
```

</td></tr>
<tr><td>Requires</td><td>MCU must have the SPI1 module.</td></tr>
<tr><td>Example</td><td>

```
/* 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);
```

</td></tr>
</table>

Spi_Read

Prototype	`unsigned Spi_Read(unsigned data_out);`
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`
	`char take, buffer;`
	`...`
	`take = Spi_Read(buffer);`

Spi_Write

Prototype	`void Spi_Write(unsigned data_out);`
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 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`
	`char buffer;`
	`...`
	`Spi_Write(buffer);`

Spi_Set_Active

Prototype	`void Spi_Set_Active(char SpiNo);`
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	`//Sets the SPI2 module active` `Spi_Set_Active(2);`

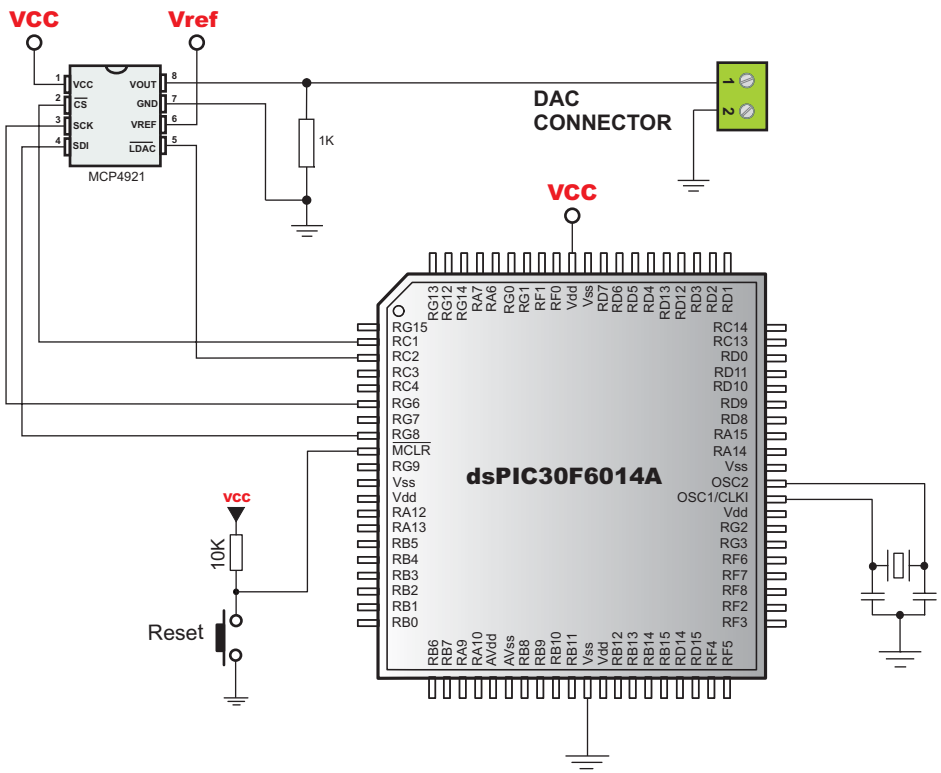
Library Example

The code demonstrates how to use SPI library functions for communication between SPI2 module of the MCU and MCP4921 DAC chip.

```
const char CS_PIN = 1;
const char LD_PIN = 2;

unsigned int value;

void InitMain() {
  ADPCFG = 0xFFFF; // Set AN pins as digital

  Spi2_Init(); // Initialize SPI2 module

  TRISC.CS_PIN = 0; // Set CS pin as output
  TRISC.LD_PIN = 0; // Set LD pin as output
} //~


// DAC increments (0..4095) --> output voltage (0..Vref)
void DAC_Output(unsigned int valueDAC) {
char temp;

PORTC.CS_PIN = 0; // Select DAC module
PORTC.LD_PIN = 0; // Enable data transfer

// Send 2 bytes of valueDAC variable
temp = (valueDAC >> 8) & 0x0F; // Prepare hi-byte for transfer
// It's a 12-bit number, so only
// lower nibble of high byte is used
temp |= 0x30; // Set MCP4921 control bits
Spi2_Write(temp); // Send data via SPI

temp = valueDAC; // Prepare lo-byte for transfer
Spi2_Write(temp); // Send data via SPI

PORTC.LD_PIN = 1; // Disable data transfer
PORTC.CS_PIN = 1; // Deselect DAC module

} //~

// continues ...
```

```
// continued ...

void main() {
InitMain();

value = 2047; // When program starts, DAC gives
              // the output in the mid-range

while (1) {  // Main loop
  DAC_Output(value++);
  if (value > 4095)
  value = 0;
  Delay_ms(10);
}
} //~!
```

Hardware Connection



SPI Ethernet Library

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

The ENC28J60 meets all 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 memory. 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 board hang or some requests might be missed.

Note: For advanced users there is a header file (`enc28j60_libprivate.h`) 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	`void Spi_Ethernet_Init(unsigned int *resetPort, unsigned int resetBit, unsigned int *CSportPtr, unsigned int CSbit, unsigned char *mac, unsigned char *ip, unsigned char fullDuplex);`
Returns	Nothing.
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	`Spi_Ethernet_Init(&PORTF, 0, &PORTF, 1, myMacAddr, myIpAddr, Spi_Ethernet_FULLDUPLEX);`

SPI_Ethernet_doPacket

Prototype	`void SPI_Ethernet_doPacket();`
Description	The function checks if a packet was received. If the packet has been received 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	`void SPI_Ethernet_putByte(unsigned char v);`
Returns	Nothing.
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	`char data;`
	`...`
	`SPI_Ethernet_putByte(data);`

SPI_Ethernet_getByte

Prototype	`unsigned char SPI_Ethernet_getByte();`
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	`char buffer;` `buffer = SPI_Ethernet_getByte();`

SPI_Ethernet_UserTCP

Prototype	`unsigned int SPI_Ethernet_UserTCP(unsigned char *remoteHost, unsigned int remotePort, unsigned int localPort, unsigned int reqLength);`
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	`unsigned int SPI_Ethernet_UserUDP(unsigned char *remoteHost, unsigned int remotePort, unsigned int destPort, unsigned int reqLength);`
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.

```
#define Spi_Ethernet_HALFDUPLEX      0
#define Spi_Ethernet_FULLDUPLEX      1

/************************************************************
 * ROM constant strings
 */
const unsigned char httpHeader[] = "HTTP/1.1 200 OK\nContent-type: " ; // HTTP header
const unsigned char httpMimeTypeHTML[] = "text/html\n\n" ;           // HTML MIME type
const unsigned char httpMimeTypeScript[] = "text/plain\n\n" ;       // TEXT MIME type
unsigned char httpMethod[] = "GET /";
/*
 * web page, split into 2 parts :
 * when coming short of ROM, fragmented data is handled more efficiently by linker
 *
 * this HTML page calls the boards to get its status, and builds itself with
javascript
 */
const char     *indexPage =
"<meta http-equiv=\"refresh\" content=\"3;url=http://192.168.20.60\">\
<HTML><HEAD></HEAD><BODY>\
<h1>dsPIC + ENC28J60 Mini Web Server</h1>\
<a href=/>Reload</a>\
<script src=/s></script>\
<table><tr><td valign=top><table border=1 style=\"font-size:20px ;font-family: termi-
nal ;\">\
<tr><th colspan=2>ADC</th></tr>\
<tr><td>AN10</td><td><script>document.write(AN10)</script></td></tr>\
</table></td><td><table border=1 style=\"font-size:20px ;font-family: terminal ;\">\
<tr><th colspan=2>PORTB</th></tr>\
<script>\
var str,i;\
str=\"\";\
for(i=0;i<8;i++)\
{ str+=\"<tr><td bgcolor=pink>BUTTON #\"+i+\"</td>\";\
if(PORTB&(1<<i)){ str+=\"<td bgcolor=red>ON\";}\
else { str+=\"<td bgcolor=#cccccc>OFF\";}\
str+=\"</td></tr>\";}\


// continues...
```

```
// continued...

document.write(str) ;\
</script>\
" ;

const char    *indexPage2 =  "</table></td><td>\
<table border=1 style=\"font-size:20px ;font-family: terminal ;\">\
<tr><th colspan=3>PORTD</th></tr>\
<script>\
var str,i;\
str=\"\";\
for(i=0;i<8;i++)\
{ str+=\"<tr><td bgcolor=yellow>LED #\"+i+\"</td>\";\
if(PORTD&(1<<i)){ str+=\"<td bgcolor=red>ON\";}\
else { str+=\"<td bgcolor=#cccccc>OFF\";}\
str+=\"</td><td><a href=/t\"+i+\">Toggle</a></td></tr>\";}\
document.write(str) ;\
</script>\
</table></td></tr></table>\
This is HTTP request #<script>document.write(REQ)</script></BODY></HTML>\
" ;

/***********************************
 * RAM variables
 */
unsigned char    myMacAddr[ 6]  = { 0x00, 0x14, 0xA5, 0x76, 0x19, 0x3f} ;// my MAC address
unsigned char    myIpAddr[ 4]  = { 192, 168, 20, 60}  ;                  // my IP address
unsigned char    getRequest[ 15]  ;                                    // HTTP request buffer
unsigned char    dyna[ 31]  ;                                  // buffer for dynamic response
unsigned long    httpCounter = 0 ;                               // counter of HTTP requests

/*****************************************
 * functions
 */

/*
 * put the constant string pointed to by s to the ENC transmit buffer
 */


//continues...
```

```
// continued ...

unsigned int     putConstString(const char *s)
         {
         unsigned int ctr = 0 ;

         while(*s)
                  {
                  Spi_Ethernet_putByte(*s++) ;
                  ctr++ ;
                  }
         return(ctr) ;
         }

/*
 * put the string pointed to by s to the ENC transmit buffer
 */
unsigned int     putString(char *s)
         {
         unsigned int ctr = 0 ;

         while(*s)
                  {
                  Spi_Ethernet_putByte(*s++) ;
                  ctr++ ;
                  }
         return(ctr) ;
         }

/*
 * this function is called by the library
 * the user accesses to the HTTP request by successive calls to
Spi_Ethernet_getByte()
 * the user puts data in the transmit buffer by successive calls to
Spi_Ethernet_putByte()
 * the function must return the length in bytes of the HTTP reply, or 0 if nothing
to transmit
 *
 * if you don't need to reply to the HTTP requests,
 * just define this function with return(0) as a single statement
 *
 */


// continues...
```

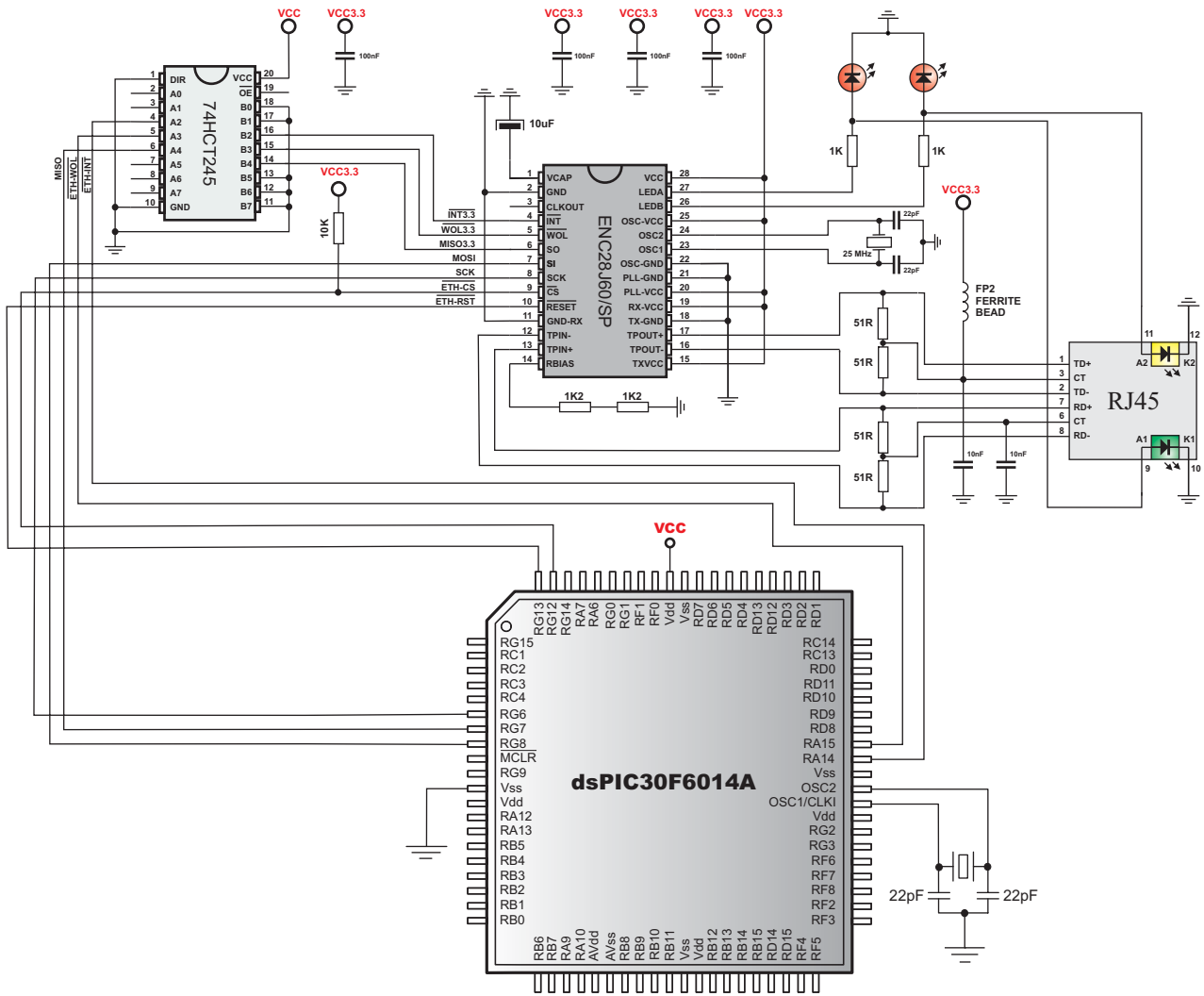
```
// continued...

unsigned int     Spi_Ethernet_UserTCP(unsigned char *remoteHost, unsigned int
remotePort, unsigned int localPort, unsigned int reqLength)
         {
         unsigned int     len = 0 ;         // my reply length
         unsigned int     i ;               // general purpose integer

         if(localPort != 80)                // I listen only to web request on port 80
                  {
                  return(0) ;
                  }

         // get 10 first bytes only of the request, the rest does not matter here
         for(i = 0 ; i < 10 ; i++)
                  {
                  getRequest[ i]  = Spi_Ethernet_getByte() ;
                  }
         getRequest[ i]  = 0 ;

         if(memcmp(getRequest, httpMethod, 5))
         // only the GET method is supported here
                  {
                  return(0) ;
                  }

         httpCounter++ ;                                  // one more request done

         if(getRequest[ 5]  == 's')
         // if request path name starts with s, store dynamic data in transmit buffer
                  {
                  // the text string replied by this request can be interpreted
                  // as javascript statements
                  // by browsers

                  len = putConstString(httpHeader) ;          // HTTP header
                  len += putConstString(httpMimeTypeScript) ; // with text MIME type

                  // add AN10 value to reply
                  intToStr(ADC_Read(10), dyna) ;
                  len += putConstString("var AN10=") ;
                  len += putString(dyna) ;
                  len += putConstString(";") ;

// continues...
```

```
// continued...

                    // add PORTB value (buttons) to reply
                    len += putConstString("var PORTB=") ;
                    intToStr(PORTB, dyna) ;
                    len += putString(dyna) ;
                    len += putConstString(";") ;

                    // add PORTD value (LEDs) to reply
                    len += putConstString("var PORTD=") ;
                    intToStr(PORTD, dyna) ;
                    len += putString(dyna) ;
                    len += putConstString(";") ;

                    // add HTTP requests counter to reply
                    intToStr(httpCounter, dyna) ;
                    len += putConstString("var REQ=") ;
                    len += putString(dyna) ;
                    len += putConstString(";") ;
                    }
          else if(getRequest[ 5]  == 't'  // if request path name starts with t,
                                          //toggle PORTD (LED) bit number that comes after
                    {
                    unsigned char   bitMask = 0 ;                   // for bit mask

                    if(isdigit(getRequest[ 6])) // if 0 <= bit number <= 9, bits 8 & 9
                                                // does not exist but does not matter
                              {
                              bitMask = getRequest[ 6]  - '0' ; // convert ASCII to integer
                              bitMask = 1 << bitMask ;        // create bit mask
                              PORTD ^= bitMask ;        // toggle PORTD with xor operator
                              }
                    }

          if(len == 0)                                  // what to do by default
                    {
                    len =  putConstString(httpHeader) ;  // HTTP header
                    len += putConstString(httpMimeTypeHTML) ; // with HTML MIME type
                    len += putConstString(indexPage) ;          // HTML page first part
                    len += putConstString(indexPage2) ;      // HTML page second part
                    }

          return(len) ; // return to the library with the number of bytes to transmit
          }


// continues...
```

```
// continued...
/*
 * this function is called by the library
 * the user accesses to the UDP request by successive calls to Spi_Ethernet_getByte()
 * the user puts data in the transmit buffer by successive calls to
 * Spi_Ethernet_putByte()
 * the function must return the length in bytes of the UDP reply, or 0 if nothing to
 * transmit
 * if you don't need to reply to the UDP requests,
 * just define this function with return(0) as a single statement
 */
unsigned int     Spi_Ethernet_UserUDP(unsigned char *remoteHost, unsigned int
remotePort, unsigned int destPort, unsigned int reqLength)
        {
        unsigned int     len ;                                  // my reply length
        unsigned char    *ptr ;                    // pointer to the dynamic buffer
        // reply is made of the remote host IP address in human-readable format
        byteToStr(remoteHost[ 0], dyna) ;                   // first IP address byte
        dyna[ 3]  = '.' ;
        byteToStr(remoteHost[ 1], dyna + 4) ;               // second
        dyna[ 7]  = '.' ;
        byteToStr(remoteHost[ 2], dyna + 8) ;               // third
        dyna[ 11]  = '.' ;
        byteToStr(remoteHost[ 3], dyna + 12) ;              // fourth
        dyna[ 15]  = ':' ;                                  // add separator
        // then remote host port number
        intToStr(remotePort, dyna + 16) ;
        dyna[ 22]  = '[ ' ;
        intToStr(destPort, dyna + 23) ;
        dyna[ 29]  = '] ' ;
        dyna[ 30]  = 0 ;
        // the total length of the request is the
        // length of the dynamic string plus the text of the request
        len = 30 + reqLength ;
        // puts the dynamic string into the transmit buffer
        ptr = dyna ;
        while(*ptr)
                {
                Spi_Ethernet_putByte(*ptr++) ;
                }
        // then puts the request string converted into
        // upper char into the transmit buffer
        while(reqLength--)
                {
                Spi_Ethernet_putByte(toupper(Spi_Ethernet_getByte())) ;
                }

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

```
/*
 * main entry
 */
void     main()
         {
         ADPCFG |= 0xFBFF;          // all digital but rb10(AN10)

         PORTB = 0 ;
         TRISB = 0xffff ;           // set PORTB as input for buttons and adc

         PORTD = 0 ;
         TRISD = 0 ;                 // set PORTD as output,

         /*
          * starts ENC28J60 with :
          * reset bit on RC0
          * CS bit on RC1
          * my MAC & IP address
          * full duplex
          */

         // for faster SPI communication use Spi1_Init_Advanced settings
         Spi_Init();

/*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_LOW, _SPI_IDLE_2_ACTIVE);*/

Spi_Ethernet_Init(&PORTF,0, &PORTF, 1, myMacAddr, myIpAddr, Spi_Ethernet_FULLDUPLEX) ;

         while(1)                                  // do forever
                  {
                  Spi_Ethernet_doPacket() ;    // process incoming Ethernet packets

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

HW Connection



SPI Graphic LCD Library

The mikroC for dsPIC30/33 and PIC24 provides a library for operating Graphic LCD 128x64 (with commonly used Samsung KS108/KS107 controller) via SPI interface.

Note: This library supports the dsPIC30 only due to the dsPIC33 and PIC24 voltage incompatibility with the Samsung KS0108 based GLCD modules.

Note: The library uses the SPI module for communication. 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	`void Spi_Glcd_Config(char DeviceAddress, unsigned int *rstport, unsigned int rstpin, unsigned int *csport, unsigned int cspin);`
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 `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_Init

Prototype	`void 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 celect` 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 `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	`void SPI_Glcd_Set_Side(char x_pos);`
Description	Selects GLCD side. Refer to the GLCD datasheet for detail explanation. Parameters : - `x_pos`: position on x-axis. Valid values: 0..127 The parameter `x_pos` specifies the GLCD side: values from 0 to 63 specify the left side, values from 64 to 127 specify the right side. **Note**: For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine.

Example	`/*The following two lines are equivalent, and both of them select the left side of GLCD:*/`
	
	`SPI_Glcd_Set_Side(0);`
	`SPI_Glcd_Set_Side(10);`

Spi_Glcd_Set_Page

Prototype	`void Spi_Glcd_Set_Page(char page);`
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. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Set_Page(5);`

Spi_Glcd_Set_X

Prototype	`void SPI_Glcd_Set_X(char x_pos);`
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 `Spi_Glcd_Init`.
Example	`Spi_Glcd_Set_X(25);`

Spi_Glcd_Read_Data

Prototype	`char Spi_Glcd_Read_Data();`
Returns	One byte from GLCD memory.
Description	Reads data from the current location of GLCD memory and moves to the next location.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine. GLCD side, x-axis position and page should be set first. See the functions `Spi_Glcd_Set_Side`, `Spi_Glcd_Set_X`, and `Spi_Glcd_Set_Page`.

Example	`char data;`
	`...`
	`data = Spi_Glcd_Read_Data();`

Spi_Glcd_Write_Data

Prototype	`void Spi_Glcd_Write_Data(char data);`
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 `Spi_Glcd_Init` routine. GLCD side, x-axis position and page should be set first. See the functions `Spi_Glcd_Set_Side`, `Spi_Glcd_Set_X`, and `Spi_Glcd_Set_Page`.

Example	`char data;`
	`...`
	`Spi_Glcd_Write_Data(data);`

Spi_Glcd_Fill

Prototype	`void Spi_Glcd_Fill(char pattern);`
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 `Spi_Glcd_Init` routine.
Example	`// Clear screen` `Spi_Glcd_Fill(0);`

Spi_Glcd_Dot

Prototype	`void Spi_Glcd_Dot(char x_pos, char y_pos, char color);`
Description	Draws a dot on GLCD at coordinates (`x_pos`, `y_pos`). Parameters : - `x_pos`: x position. Valid values: 0..127 - `y_pos`: y position. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state. **Note**: For x and y axis layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine.
Example	`// Invert the dot in the upper left corner` `Spi_Glcd_Dot(0, 0, 2);`

Spi_Glcd_Line

Prototype	`void SPI_Glcd_Line(int x_start, int y_start, int x_end, int y_end, char color);`
Description	Draws a line on GLCD. Parameters : - `x_start`: x coordinate of the line start. Valid values: 0..127 - `y_start`: y coordinate of the line start. Valid values: 0..63 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 Parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine.
Example	`// Draw a line between dots (0,0) and (20,30)` `Spi_Glcd_Line(0, 0, 20, 30, 1);`

Spi_Glcd_V_Line

Prototype	`void Spi_Glcd_V_Line(char y_start, char y_end, char x_pos, char color);`
Description	Draws a vertical line on GLCD. Parameters : - `y_start`: y coordinate of the line start. Valid values: 0..63 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `x_pos`: x coordinate of vertical line. Valid values: 0..127 - `color`: color parameter. Valid values: 0..2 Parameter `color` determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine.
Example	`// Draw a vertical line between dots (10,5) and (10,25)` `Spi_Glcd_V_Line(5, 25, 10, 1);`

Spi_Glcd_H_Line

Prototype	`void Spi_Glcd_H_Line(char x_start, char x_end, char y_pos, char color);`
Description	Draws a horizontal line on GLCD. Parameters : - `x_start`: x coordinate of the line start. Valid values: 0..127 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_pos`: y coordinate of horizontal line. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine.
Example	`// Draw a horizontal line between dots (10,20) and (50,20)` `Spi_Glcd_H_Line(10, 50, 20, 1);`

Spi_Glcd_Rectangle

Prototype	`void Spi_Glcd_Rectangle(char x_upper_left, char y_upper_left, char x_bottom_right, char y_bottom_right, char color);`
Description	Draws a rectangle on GLCD. Parameters : - `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right rectangle corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine.
Example	`// Draw a rectangle between dots (5,5) and (40,40)` `Spi_Glcd_Rectangle(5, 5, 40, 40, 1);`

Spi_Glcd_Box

Prototype	`void Spi_Glcd_Box(char x_upper_left, char y_upper_left, char x_bottom_right, char y_bottom_right, char color);`
Description	Draws a box on GLCD. Parameters : - `x_upper_left`: x coordinate of the upper left box corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left box corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right box corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right box corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the color of the box fill: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine.
Example	`// Draw a box between dots (5,15) and (20,40)` `Spi_Glcd_Box(5, 15, 20, 40, 1);`

Spi_Glcd_Circle

Prototype	`void Spi_Glcd_Circle(int x_center, int y_center, int radius, char color);`
Description	Draws a circle on GLCD. Parameters : - `x_center`: x coordinate of the circle center. Valid values: 0..127 - `y_center`: y coordinate of the circle center. Valid values: 0..63 - `radius`: radius size - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the circle line: 0 white, 1 black, and 2 inverts each dot.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine.
Example	`// Draw a circle with center in (50,50) and radius=10` `Spi_Glcd_Circle(50, 50, 10, 1);`

Spi_Glcd_Set_Font

Prototype	`void SPI_Glcd_Set_Font(const char *activeFont, char aFontWidth, char aFontHeight, unsigned int aFontOffs);`
Description	Sets font that will be used with Spi_Glcd_Write_Char and Spi_Glcd_Write_Text routines. Parameters : - `activeFont`: font to be set. Needs to be formatted as an array of char - `aFontWidth`: width of the font characters in dots. - `aFontHeight`: height of the font characters in dots. - `aFontOffs`: number that represents difference between the mikroC character set and regular ASCII set (eg. if 'A' is 65 in ASCII character, and 'A' is 45 in the mikroC 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.c`" file located in the Uses folder or create his own fonts.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine.
Example	`// Use the custom 5x7 font "myfont" which starts with space (32):` `Spi_Glcd_Set_Font(myfont, 5, 7, 32);`

Spi_Glcd_Write_Char

Prototype	`void SPI_Glcd_Write_Char(char chr1, char x_pos, char page_num, char color);`
Description	Prints character on GLCD. Parameters : - `chr1`: character to be written - `x_pos`: character starting position on x-axis. Valid values: 0..(127-FontWidth) - `page_num`: the number of the page on which character will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the character: 0 white, 1 black, and 2 inverts each dot. **Note**: For x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine. Use the `Spi_Glcd_Set_Font` to specify the font for display; if no font is specified, then the default 5x8 font supplied with the library will be used.
Example	`// Write character 'C' on the position 10 inside the page 2:` `Spi_Glcd_Write_Char('C', 10, 2, 1);`

Spi_Glcd_Write_Text

Prototype	`void SPI_Glcd_Write_Text(char text[], char x_pos, char page_num, char color);`
Description	Prints text on GLCD. Parameters : - `text`: text to be written - `x_pos`: text starting position on x-axis. - `page_num`: the number of the page on which text will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the text: 0 white, 1 black, and 2 inverts each dot. **Note**: For x axis and page layout explanation see schematic at the bottom of this page.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine. Use the `Spi_Glcd_Set_Font` to specify the font for display; if no font is specified, then the default 5x8 font supplied with the library will be used.
Example	`//Write text "Hello world!" on the position 10 inside the page 2:` `Spi_Glcd_Write_Text("Hello world!", 10, 2, 1);`

Spi_Glcd_Image

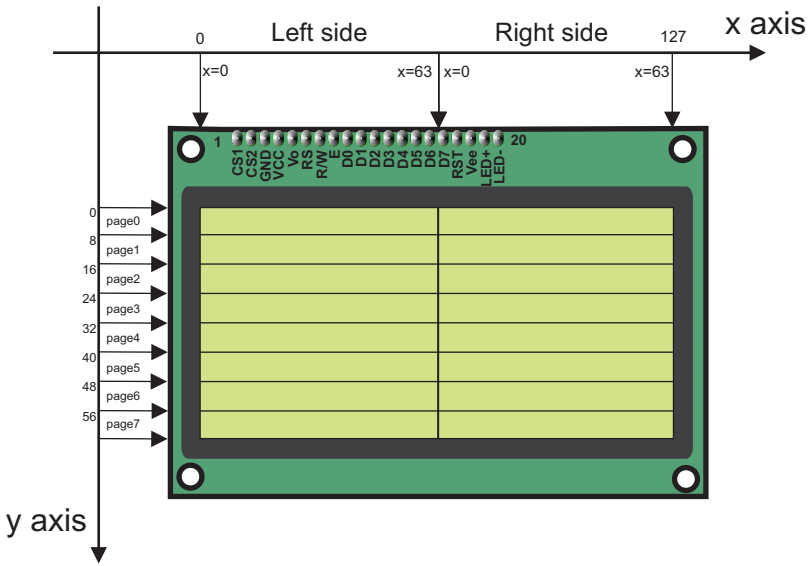
Prototype	`void Spi_Glcd_Image(const char *image);`
Description	Displays bitmap on GLCD. Parameters : - `image`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroC for dsPIC30/33 and PIC24 pointer to const and pointer to RAM equivalency). Use the mikroC's integrated GLCD Bitmap Editor (menu option **Tools › GLCD Bitmap Editor**) to convert image to a constant array suitable for displaying on GLCD.
Requires	GLCD needs to be initialized for SPI communication, see `Spi_Glcd_Init` routine.
Example	`// Draw image my_image on GLCD` `Spi_Glcd_Image(my_image);`

Library Example

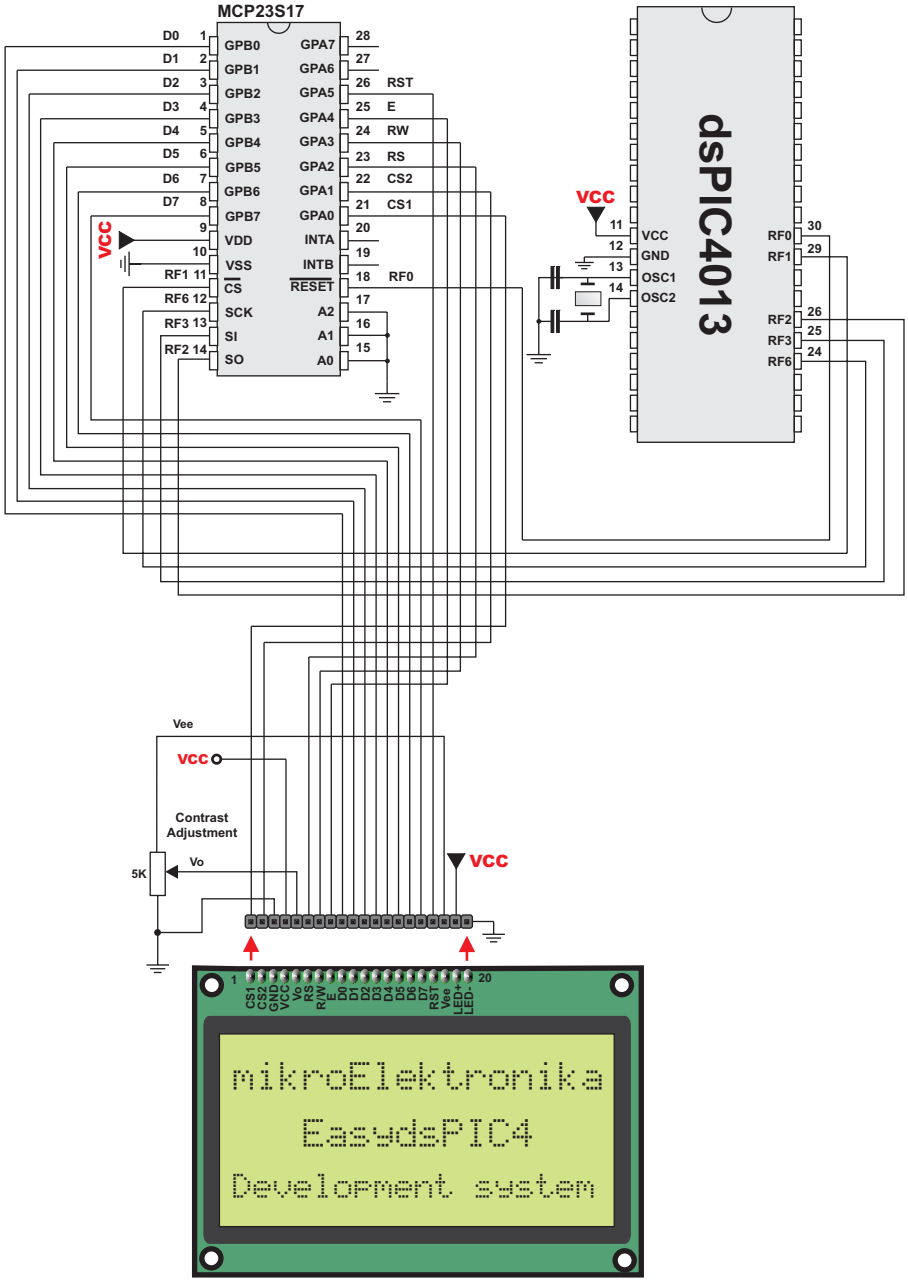
The example demonstrates how to communicate to KS0108 GLCD via the SPI module, using serial to parallel convertor MCP23S17.

```
extern const unsigned short
  truck_bmp[ ] ;

char ii;
unsigned int jj;
char *someText;

void delay2S() {
  Delay_ms(2000);
}

void main() {
  ADPCFG |= 0xFF;
  // for faster spi communication use Spi_Init_Advanced function
  Spi_Init();
  Spi_Glcd_Init(0, &PORTF, 0, &PORTF, 1);
  Spi_Glcd_Fill(0xAA);
  delay2S();
  while(1) {
       Spi_Glcd_Image( truck_bmp );
       delay2S();
       Spi_Glcd_Fill(0x00);

       for(jj = 1; jj <= 40; jj++)
         Spi_Glcd_Dot(jj,jj,1);
       delay2S();

       Spi_Glcd_Fill(0x00);
       Spi_Glcd_Line(120, 1, 5,60, 1);
       delay2S();
       Spi_Glcd_Line(12, 42, 5,60, 1);
       delay2S();

       Spi_Glcd_Rectangle(12, 20, 93,57, 1);
       delay2S();


//continues ...
```

```
        //continued..

        Spi_Glcd_Line(120, 12, 12,60, 1);
        delay2S();

        Spi_Glcd_H_Line(5, 40, 6, 1);
        delay2S();
        Spi_Glcd_Line(0, 12, 120, 60, 1);
        Spi_Glcd_V_Line(7, 63, 127, 1);
        delay2S();

        for(ii = 1; ii <= 10; ii++)
          Spi_Glcd_Circle(63, 32, 3*ii, 1);

        delay2S();
        Spi_Glcd_Box(12, 20, 70, 57, 2);
        delay2S();

        Spi_Glcd_Fill(0x00);

        Spi_Glcd_Set_Font(&System3x6, 3, 6, 32);
        someText = "SMALL FONT: 3X6";
        Spi_Glcd_Write_Text(someText, 20, 5, 1);

        Spi_Glcd_Set_Font(&FontSystem5x8, 5, 8, 32);
        someText = "Large Font 5x8";
        Spi_Glcd_Write_Text(someText, 3, 4, 1);
        delay2S();
    }
} //~!
```



Hardware Connection



SPI LCD Library (4-bit interface)

The mikroC for dsPIC30/33 and PIC24 provides a library for communication with LCD (with HD44780 compliant controllers) in 4-bit mode via SPI interface.

Note: This library supports the dsPIC30 only due to the dsPIC33 and PIC24 voltage incompatibility with the LCD modules.

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 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	`void Spi_Lcd_Config(char DeviceAddress, unsigned int *rstport, unsigned int rstpin, unsigned int *csport, unsigned int cspin);`
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 `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	`void 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 celect` 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 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	`void Spi_Lcd_Out(char row, char column, char *text);`
Description	Prints text on the LCD starting from specified position. Both string variables and literals can be passed as a text. Parameters : - `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Requires	LCD needs to be initialized for SPI communication, see `Spi_Lcd_Config` 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	`void Spi_Lcd_Out_CP(char *text);`
Description	Prints text on the LCD at current cursor position. Both string variables and literals can be passed as a text. Parameters : - `text`: text to be written
Requires	LCD needs to be initialized for SPI communication, see `Spi_Lcd_Config` and `Spi_Lcd_Init` routines.
Example	`// Write text "Here!" at current cursor position:` `Spi_Lcd_Out_CP("Here!");`

Spi_Lcd_Chr

Prototype	`void Spi_Lcd_Chr(char Row, char Column, char Out_Char);`
Description	Prints character on LCD at specified position. Both variables and literals can be passed as character. Parameters : - `Row`: writing position row number - `Column`: writing position column number - `Out_Char`: character to be written
Requires	LCD needs to be initialized for SPI communication, see `Spi_Lcd_Config` 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	`void Spi_Lcd_Chr_CP(char Out_Char);`
Description	Prints character on LCD at current cursor position. Both variables and literals can be passed as character. Parameters : - `Out_Char`: character to be written
Requires	LCD needs to be initialized for SPI communication, see `Spi_Lcd_Config` and `Spi_Lcd_Init` routines.
Example	`// Write character "e" at current cursor position:` `Spi_Lcd_Chr_Cp('e');`

Spi_Lcd_Cmd

Prototype	`void Spi_Lcd_Cmd(char out_char);`
Description	Sends command to LCD. Parameters : - `out_char`: command to be sent **Note**: Predefined constants can be passed to the function, see Available LCD Commands.
Requires	LCD needs to be initialized for SPI communication, see `Spi_Lcd_Config` and `Spi_Lcd_Init` routines.
Example	`// Clear LCD display:` `Spi_Lcd_Cmd(LCD_CLEAR);`

LCD Commands

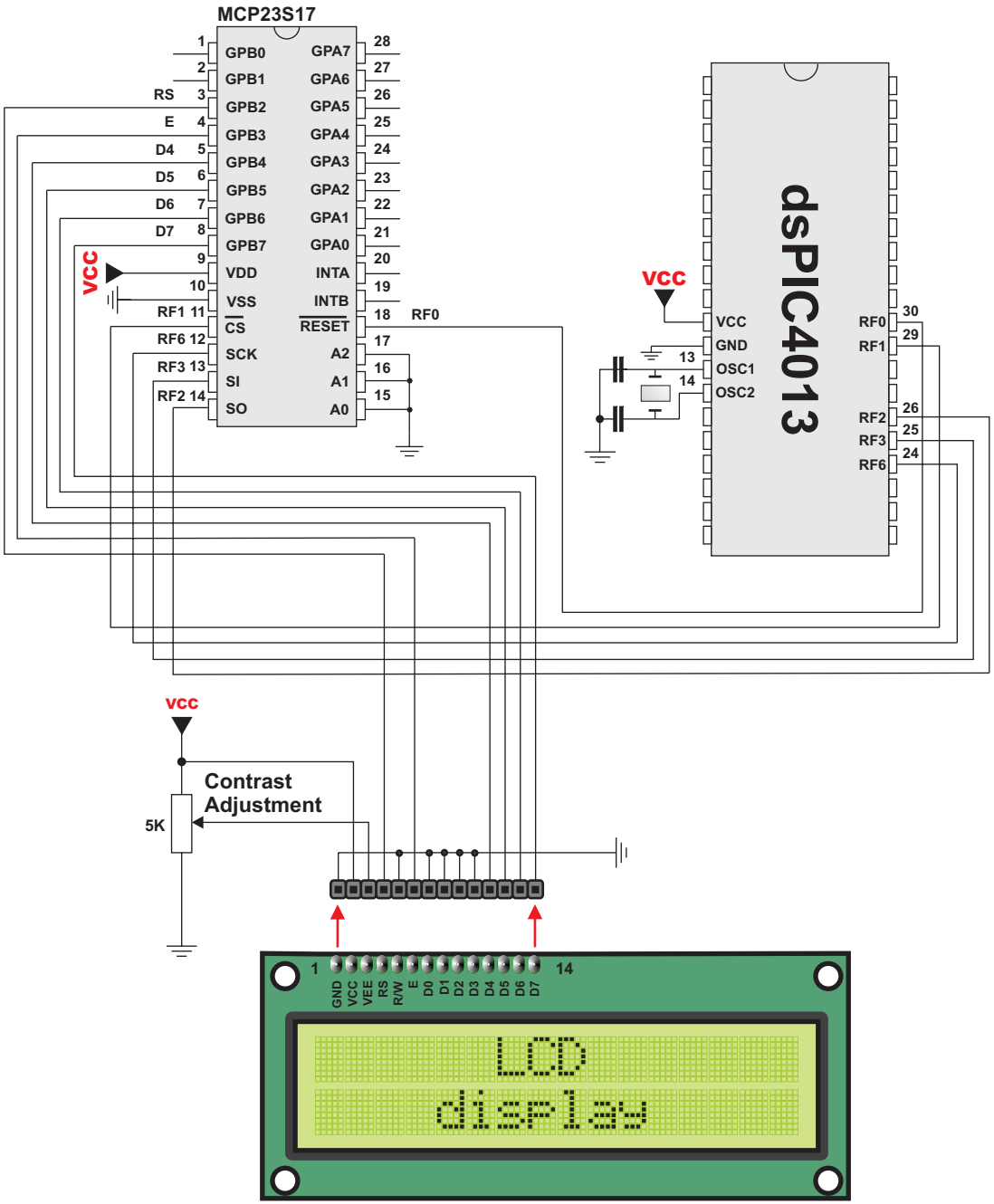
LCD Command	Purpose
LCD_FIRST_ROW	Move cursor to 1st row
LCD_SECOND_ROW	Move cursor to 2nd row
LCD_THIRD_ROW	Move cursor to 3rd row
LCD_FOURTH_ROW	Move cursor to 4th row
LCD_CLEAR	Clear display
LCD_RETURN_HOME	Return cursor to home position, returns a shifted display to original position. Display data 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 (default pin settings)

Use Spi_Lcd_Init for default pin settings (see the first figure below).

```
char *text = "mikroElektronika";

void main() {
  Spi1_Init();                    // initializes spi1 by default
  Spi_Lcd_Init();                 // initialize lcd over spi interface
  Spi_Lcd_Cmd(LCD_CLEAR);         // Clear display
  Spi_Lcd_Cmd(LCD_CURSOR_OFF);    // Turn cursor off
  Spi_Lcd_Out(1,6, "mikroE");     // Print text to LCD, 1st row, 6th column
  Spi_Lcd_Chr_CP('!');            // append !
  Spi_Lcd_Out(2,1, text);         // Print text to LCD, 2nd row, 1st column
  Spi_Lcd_Out(3,1,"mikroE");      // for lcd with more than two rows
  Spi_Lcd_Out(4,15,"mikroE");     // for lcd with more than two rows
} //~!
```

Hardware Connection



SPI LCD8 (8-bit interface) Library

The mikroC for dsPIC30/33 and PIC24 provides a library for communication with LCD (with HD44780 compliant controllers) in 8-bit mode via SPI interface.

Note: This library supports the dsPIC30 only due to the dsPIC33 and PIC24 voltage incompatibility with the LCD modules.

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 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	`void Spi_Lcd8_Config(char DeviceAddress, unsigned int *rstport, unsigned int rstpin, unsigned int *csport, unsigned int cspin);`
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 `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	`void 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 celect` 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 `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	`void Spi_Lcd8_Out(unsigned short row, unsigned short column, char *text);`
Description	Prints text on LCD starting from specified position. Both string variables and literals can be passed as a text. Parameters : - `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Requires	LCD needs to be initialized for SPI communication, see `Spi_Lcd8_Config` 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	`void Spi_Lcd8_Out_CP(char *text);`
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 `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	`void Spi_Lcd8_Chr(unsigned short row, unsigned short column, char out_char);`
Description	Prints character on LCD at specified position. Both variables and literals can be passed as character. Parameters : - `row`: writing position row number - `column`: writing position column number - `out_char`: character to be written
Requires	LCD needs to be initialized for SPI communication, see `Spi_Lcd8_Config` 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	`void Spi_Lcd8_Chr_CP(char out_char);`
Description	Prints character on LCD at current cursor position. Both variables and literals can be passed as character. Parameters : - `out_char`: character to be written
Requires	LCD needs to be initialized for SPI communication, see `Spi_Lcd8_Config` and `Spi_Lcd8_Init` routines.
Example	`// Write character "e" at current cursor position:` `Spi_Lcd8_Chr_Cp('e');`

Spi_Lcd8_Cmd

Prototype	`void Spi_Lcd8_Cmd(char out_char);`
Description	Sends command to LCD. Parameters : - `out_char`: command to be sent **Note**: Predefined constants can be passed to the function, see Available LCD Commands.
Requires	LCD needs to be initialized for SPI communication, see `Spi_Lcd8_Config` and `Spi_Lcd8_Init` routines.
Example	`// Clear LCD display:` `Spi_Lcd8_Cmd(LCD_CLEAR);`

LCD Commands

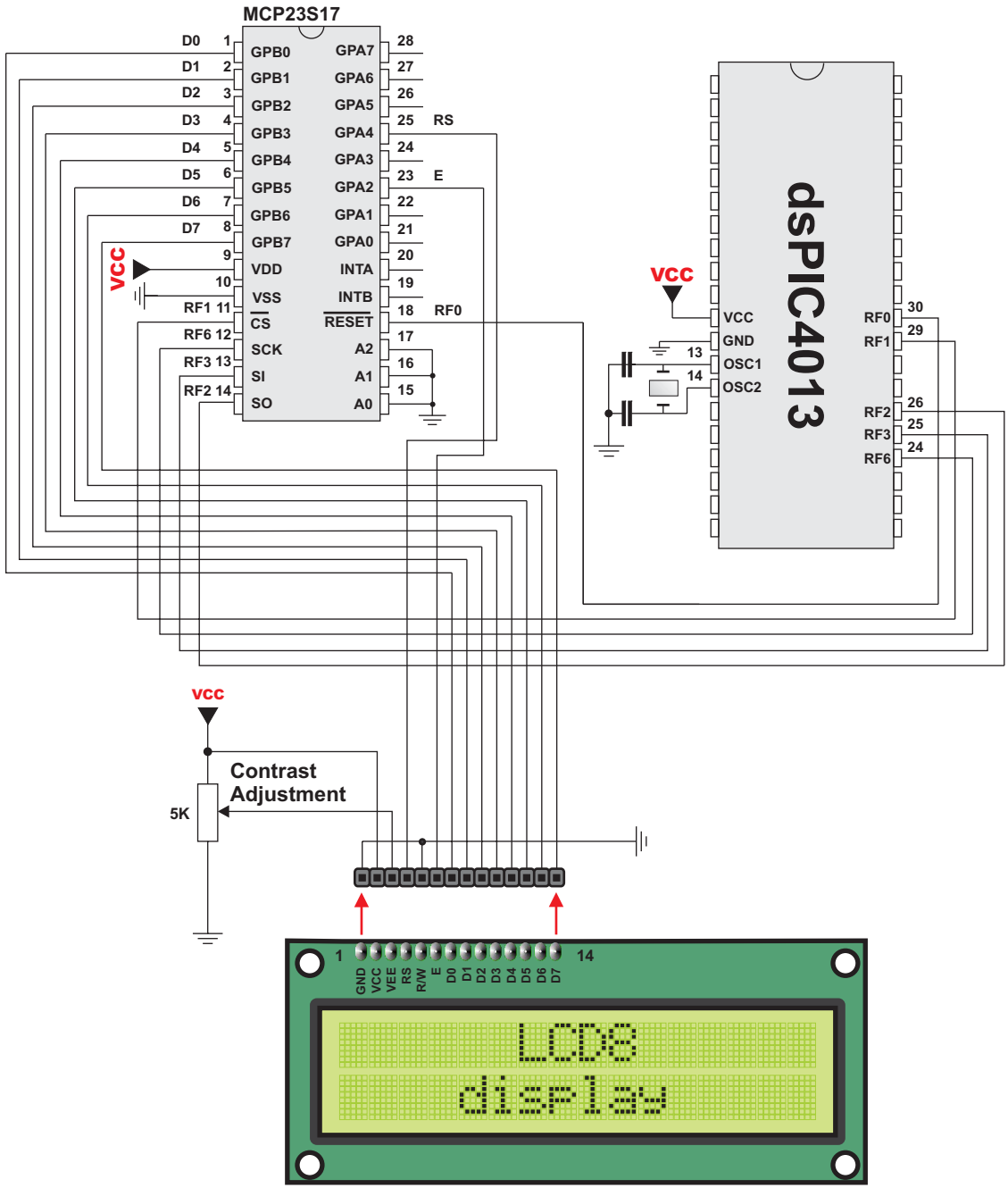
LCD Command	Purpose
LCD_FIRST_ROW	Move cursor to 1st row
LCD_SECOND_ROW	Move cursor to 2nd row
LCD_THIRD_ROW	Move cursor to 3rd row
LCD_FOURTH_ROW	Move cursor to 4th row
LCD_CLEAR	Clear display
LCD_RETURN_HOME	Return cursor to home position, returns a shifted display to original position. Display data 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 (default pin settings)

Use Spi_Lcd8_Init for default pin settings (see the first figure below).

```
char *text = "mikroE";
void main() {
  Spi1_Init();                          // initialize spi interface
  Spi_Lcd8_Init();                      // intialize lcd in 8bit mode via spi
  Spi_Lcd8_Cmd(LCD_CLEAR);              // Clear display
  Spi_Lcd8_Cmd(LCD_CURSOR_OFF);         // Turn cursor off
  Spi_Lcd8_Out(1,6, text);              // Print text to LCD, 1st row, 6th column...
  Spi_Lcd8_Chr_CP('!');                 // append '!'
  Spi_Lcd8_Out(2,1, "mikroelektronika"); // Print text to LCD, 2nd row, 1st column...
  Spi_Lcd8_Out(3,1, text);              // for lcd modules with more than two rows
  Spi_Lcd8_Out(4,15, text);             // for lcd modules with more than two rows
} //~!
```

Hardware Connection



SPI T6963C Graphic LCD Library

The mikroC 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.

Note: This library supports the dsPIC30 only due to the dsPIC33 and PIC24 voltage incompatibility with a certain T6963C based GLCD modules.

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

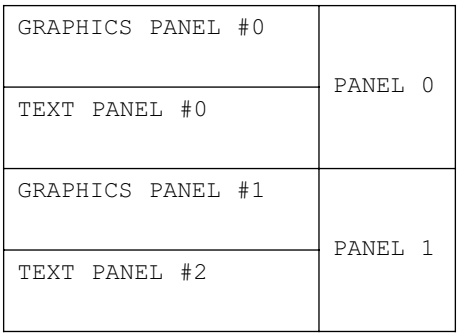
Library Routines

```
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
```

Note: The following low level library routines are implemented as macros. These macros can be found in the `Spi_T6963C.h` header file which is located in the SPI T6963C example projects folders.

```
Spi_T6963C_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	`void Spi_T6963C_Init(unsigned int width, unsigned int height, unsigned int fntW, char DeviceAddress, unsigned int *rstport, unsigned int rstpin, unsigned int *csport, unsigned int cspin, unsigned char wr, unsigned char rd, unsigned char cd, unsigned char rst);`
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). schematic:  *//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	`Spi_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 PORTF.0`
	`* chip select pin on PORTF.1`
	`* 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 the library`
	`* device address is 0`
	`*/`

Spi_T6963C_writeData

Prototype	`void Spi_T6963C_writeData(unsigned char data);`
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	`void Spi_T6963C_writeCommand(unsigned char data);`
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(Spi_T6963C_CURSOR_POINTER_SET);`

Spi_T6963C_setPtr

Prototype	`void Spi_T6963C_setPtr(unsigned int p, unsigned char c);`
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	`void 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	`void Spi_T6963C_fill(unsigned char v, unsigned int start, unsigned int len);`
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	`void Spi_T6963C_dot(int x, int y, unsigned char color);`
Description	Draws a dot in the current graphic panel of GLCD at coordinates (x, y). Parameters : - `x`: dot position on x-axis - `y`: dot position on y-axis - `color`: color parameter. Valid values: Spi_T6963C_BLACK and Spi_T6963C_WHITE
Requires	Toshiba GLCD module needs to be initialized. See the `Spi_T6963C_init` routine.
Example	`Spi_T6963C_dot(x0, y0, pcolor);`

Spi_T6963C_write_char

Prototype	`void Spi_T6963C_write_char(unsigned char c, unsigned char x, unsigned char y, unsigned char mode);`
Description	Writes a char in the current text panel of GLCD at coordinates (x, y). Parameters : - `c`: char to be written - `x`: char position on x-axis - `y`: char position on y-axis - `mode`: mode parameter. Valid values: `Spi_T6963C_ROM_MODE_OR`, `Spi_T6963C_ROM_MODE_XOR`, `Spi_T6963C_ROM_MODE_AND` and `Spi_T6963C_ROM_MODE_TEXT` Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical “exclusive OR”. This can be useful to display text in negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba GLCD module needs to be initialized. See the `Spi_T6963C_init` routine.
Example	`Spi_T6963C_write_char("A",22,23,AND);`

Spi_T6963C_write_text

Prototype	`void Spi_T6963C_write_text(unsigned char *str, unsigned char x, unsigned char y, unsigned char mode);`
Description	Writes text in the current text panel of GLCD at coordinates (x, y). Parameters : - `str`: text to be written - `x`: text position on x-axis - `y`: text position on y-axis - `mode`: mode parameter. Valid values: `Spi_T6963C_ROM_MODE_OR`, `Spi_T6963C_ROM_MODE_XOR`, `Spi_T6963C_ROM_MODE_AND` and `Spi_T6963C_ROM_MODE_TEXT` Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical “exclusive OR”. This can be useful to display text in negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on the display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba GLCD module needs to be initialized. See the `Spi_T6963C_init` routine.
Example	`Spi_T6963C_write_text("GLCD LIBRARY DEMO, WELCOME !", 0, 0, T6963C_ROM_MODE_EXOR);`

Spi_T6963C_line

Prototype	`void Spi_T6963C_line(int x0, int y0, int x1, int y1, unsigned char pcolor);`
Description	Draws a line from (x0, y0) to (x1, y1). Parameters : - `x0`: x coordinate of the line start - `y0`: y coordinate of the line end - `x1`: x coordinate of the line start - `y1`: y coordinate of the line end - `pcolor`: color parameter. Valid values: `Spi_T6963C_BLACK` and `Spi_T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the `Spi_T6963C_init` routine.
Example	`Spi_T6963C_line(0, 0, 239, 127, T6963C_WHITE);`

Spi_T6963C_rectangle

Prototype	`void Spi_T6963C_rectangle(int x0, int y0, int x1, int y1, unsigned char pcolor);`
Description	Draws a rectangle on GLCD. Parameters : - `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `pcolor`: color parameter. Valid values: `Spi_T6963C_BLACK` and `Spi_T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the `Spi_T6963C_init` routine.
Example	`Spi_T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE);`

Spi_T6963C_box

Prototype	`void Spi_T6963C_box(int x0, int y0, int x1, int y1, unsigned char pcolor);`
Description	Draws a box on the GLCD Parameters : - `x0`: x coordinate of the upper left box corner - `y0`: y coordinate of the upper left box corner - `x1`: x coordinate of the lower right box corner - `y1`: y coordinate of the lower right box corner - `pcolor`: color parameter. Valid values: `Spi_T6963C_BLACK` and `Spi_T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the `Spi_T6963C_init` routine.
Example	`Spi_T6963C_box(0, 119, 239, 127, T6963C_WHITE);`

Spi_T6963C_circle

Prototype	`void Spi_T6963C_circle(int x, int y, long r, unsigned char pcolor);`
Description	Draws a circle on the GLCD. Parameters : - `x`: x coordinate of the circle center - `y`: y coordinate of the circle center - `r`: radius size - `pcolor`: color parameter. Valid values: `Spi_T6963C_BLACK` and `Spi_T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the `Spi_T6963C_init` routine.
Example	`Spi_T6963C_circle(120, 64, 110, T6963C_WHITE);`

Spi_T6963C_image

Prototype	`void Spi_T6963C_image(const char *pic);`
Description	Displays bitmap on GLCD. Parameters : - `pic`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroC for dsPIC30 pointer to const and pointer to RAM equivalency). Use the mikroC'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	`void Spi_T6963C_sprite(unsigned char px, unsigned char py, const char *pic, unsigned char sx, unsigned char sy);`
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	`void Spi_T6963C_set_cursor(unsigned char x, unsigned char y);`
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	`void Spi_T6963C_clearBit(unsigned int b);`
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	`void Spi_T6963C_setBit(unsigned int b);`
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	`void Spi_T6963C_negBit(unsigned int b);`
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	`void Spi_T6963C_displayGrPanel(unsigned int n);`
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	`void Spi_T6963C_displayTxtPanel(unsigned int n);`
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	`void Spi_T6963C_setGrPanel(unsigned int n);`
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	`void Spi_T6963C_setTxtPanel(unsigned int n);`
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	`void Spi_T6963C_panelFill(unsigned char v);`
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	`void Spi_T6963C_grFill(unsigned char v);`
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	`void Spi_T6963C_txtFill(unsigned char v);`
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	`void Spi_T6963C_cursor_height(unsigned char n);`
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	`void Spi_T6963C_graphics(unsigned int n);`
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	`void Spi_T6963C_text(unsigned int n);`
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	`void Spi_T6963C_cursor(unsigned int n);`
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	`void Spi_T6963C_cursor_blink(unsigned int n);`
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. Hardware configurations in this example are made for the EasydsPIC3 board and dsPIC30F4013.

```
#include          "Spi_T6963C.h"

/*
 * bitmap pictures stored in ROM
 */

extern const char mc[] ;
extern const char einstein[] ;

void main(void)
        {
        unsigned char    panel ;        // current panel
        unsigned int     i ;           // general purpose register
        unsigned char    curs ;           // cursor visibility
        unsigned int     cposx, cposy ;  // cursor x-y position

        ADPCFG = 0xFFFF;
        PORTB = 0 ;
        TRISB = 0xFF ;



//continues...
```

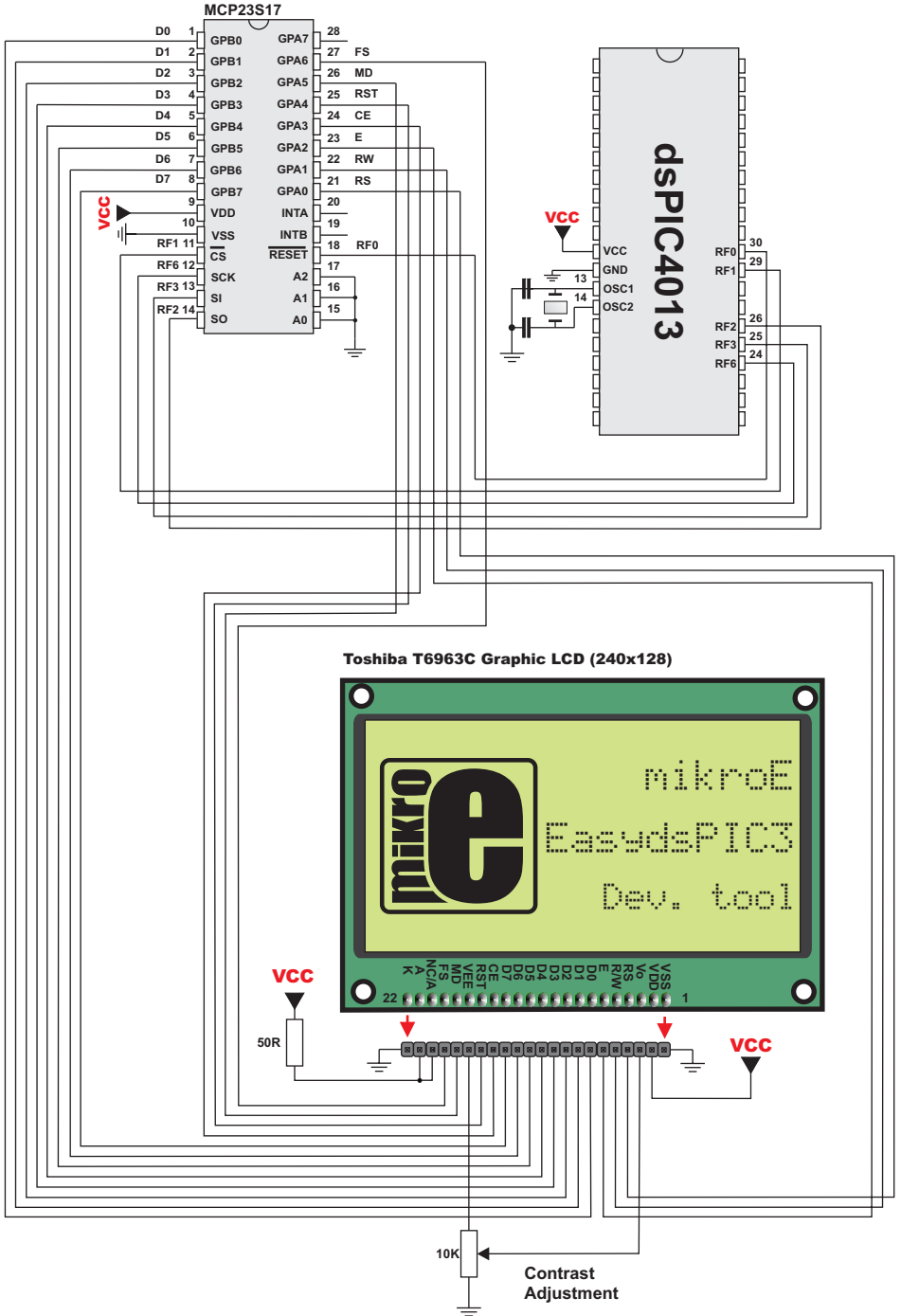
```
//continued...

     /*
      * 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 the faster spi use Spi_Init_Advanced function
        Spi_Init();
        Spi_T6963C_Init_240x128();
        /*
         * enable both graphics and text display at the same time
         */
        Spi_T6963C_graphics(1) ;
        Spi_T6963C_text(1) ;

        panel = 0 ;
        i = 0 ;
        curs = 0 ;
        cposx = cposy = 0 ;

        /*
         * text messages
         */
        Spi_T6963C_write_text(" GLCD LIBRARY DEMO, WELCOME !", 0,
0, Spi_T6963C_ROM_MODE_XOR) ;
        Spi_T6963C_write_text(" EINSTEIN WOULD HAVE LIKED mC", 0,
15, Spi_T6963C_ROM_MODE_XOR) ;

        /*
         * cursor
         */
        Spi_T6963C_cursor_height(8) ;    // 8 pixel height
        Spi_T6963C_set_cursor(0, 0) ;    // move cursor to top left
        Spi_T6963C_cursor(0) ;           // cursor off

//continues...
```

```
//continues...


        /*
         * draw rectangles
         */
        Spi_T6963C_rectangle(0, 0, 239, 127, Spi_T6963C_WHITE) ;
        Spi_T6963C_rectangle(20, 20, 219, 107, Spi_T6963C_WHITE) ;
        Spi_T6963C_rectangle(40, 40, 199, 87, Spi_T6963C_WHITE) ;
        Spi_T6963C_rectangle(60, 60, 179, 67, Spi_T6963C_WHITE) ;

        /*
         * draw a cross
         */
        Spi_T6963C_line(0, 0, 239, 127, Spi_T6963C_WHITE) ;
        Spi_T6963C_line(0, 127, 239, 0, Spi_T6963C_WHITE) ;

        /*
         * draw solid boxes
         */
        Spi_T6963C_box(0, 0, 239, 8, Spi_T6963C_WHITE) ;
        Spi_T6963C_box(0, 119, 239, 127, Spi_T6963C_WHITE) ;

        /*
         * draw circles
         */
        Spi_T6963C_circle(120, 64, 10, Spi_T6963C_WHITE) ;
        Spi_T6963C_circle(120, 64, 30, Spi_T6963C_WHITE) ;
        Spi_T6963C_circle(120, 64, 50, Spi_T6963C_WHITE) ;
        Spi_T6963C_circle(120, 64, 70, Spi_T6963C_WHITE) ;
        Spi_T6963C_circle(120, 64, 90, Spi_T6963C_WHITE) ;
        Spi_T6963C_circle(120, 64, 110, Spi_T6963C_WHITE) ;
        Spi_T6963C_circle(120, 64, 130, Spi_T6963C_WHITE) ;

        Spi_T6963C_sprite(76, 4, einstein, 88, 119) ;
        // draw a sprite

        Spi_T6963C_setGrPanel(1) ; // select other graphic panel

        Spi_T6963C_image(mc);
        //fill the graphic screen with a picture


//continued...
```

```
//continues...

        for(;;)
                {
                /*
                 * if RB1 is pressed, toggle the display
                 * between graphic panel 0 and graphic 1
                 */
                if(PORTB & 0b00000010)
                        {
                        panel++ ;
                        panel &= 1 ;
                        Spi_T6963C_displayGrPanel(panel) ;
                        Delay_ms(300) ;
                        }
                /*
                 * if RB2 is pressed, display only graphic panel
                 */
                else if(PORTB & 0b00000100)
                        {
                        Spi_T6963C_graphics(1) ;
                        Spi_T6963C_text(0) ;
                        Delay_ms(300) ;
                        }
                /*
                 * if RB3 is pressed, display only text panel
                 */
                else if(PORTB & 0b00001000)
                        {
                        Spi_T6963C_graphics(0) ;
                        Spi_T6963C_text(1) ;
                        Delay_ms(300) ;
                        }
                /*
                 * if RB4 is pressed, display text and graphic
                 * panels
                 */
                else if(PORTB & 0b00010000)
                        {
                        Spi_T6963C_graphics(1) ;
                        Spi_T6963C_text(1) ;
                        Delay_ms(300) ;
                        }



//continued...
```

```
//continues...
                        /*
                         * if RB5 is pressed, change cursor
                         */
                        else if(PORTB & 0b00100000)
                                {
                                curs++ ;
                                if(curs == 3) curs = 0 ;
                                switch(curs)
                                        {
                                        case 0:
                                                // no cursor
                                                Spi_T6963C_cursor(0) ;
                                                break ;
                                        case 1:
                                                // blinking cursor
                                                Spi_T6963C_cursor(1) ;
                                           Spi_T6963C_cursor_blink(1) ;
                                                break ;
                                        case 2:
                                                // non blinking cursor
                                                Spi_T6963C_cursor(1);
                                          Spi_T6963C_cursor_blink(0) ;
                                                break ;
                                        }
                                Delay_ms(300) ;
                                }

                        /*
                         * move cursor, even if not visible
                         */
                        cposx++ ;
                        if(cposx == Spi_T6963C_txtCols)
                                {
                                cposx = 0 ;
                                cposy++ ;
                                        if(cposy == Spi_T6963C_grHeight /
                                             Spi_T6963C_CHARACTER_HEIGHT)
                                         {
                                         cposy = 0 ;
                                         }
                                }
                        Spi_T6963C_set_cursor(cposx, cposy) ;

                        Delay_ms(100) ;
                        }
                } //~!
```

Hardware Connection



T6963C Graphic LCD Library

The mikroC 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.

Note: This library supports the dsPIC30 only due to the dsPIC33 and PIC24 voltage incompatibility with a certain T6963C based GLCD modules.

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
```

Note: The following low level library routines are implemented as macros. These macros can be found in the `T6963C.h` header file which is located in the T6963C example projects folders.

```
T6963C_clearBit
T6963C_setBit
T6963C_negBit
T6963C_displayGrPanel
T6963C_displayTxtPanel
T6963C_setGrPanel
T6963C_setTxtPanel
T6963C_panelFill
T6963C_grFill
T6963C_txtFill
T6963C_cursor_height
T6963C_graphics
T6963C_text
T6963C_cursor
T6963C_cursor_blink
```

T6963C_init

Prototype	`void T6963C_init(unsigned int width, unsigned int height, unsigned int fntW, unsigned int *data, unsigned int *cntrl, unsigned int wr, unsigned int rd, unsigned int cd, unsigned int rst);`
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). schematic: 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 ...	
Requires	Nothing.

Example	`T6963C_init(240, 128, 8, &PORTF, &PORTD, 5, 7, 6, 4) ;`
	`/*`
	`* init display for 240 pixel width and 128 pixel height`
	`* 8 bits character width`
	`* data bus on PORTF`
	`* control bus on PORTD`
	`* bit 5 is !WR`
	`* bit 7 is !RD`
	`* bit 6 is C!D`
	`* bit 4 is RST`
	`*/`

T6963C_writeData

Prototype	`void T6963C_writeData(unsigned char data);`
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	`void T6963C_writeCommand(unsigned char data);`
Description	Writes command to 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	`void T6963C_setPtr(unsigned int p, unsigned char c);`
Description	Sets the memory pointer p for command c. Parameters : - `p`: address where command should be written - `c`: command to be written
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`T6963C_setPtr(T6963C_grHomeAddr + start, T6963C_ADDRESS_POINTER_SET);`

T6963C_waitReady

Prototype	`void T6963C_waitReady(void);`
Description	Pools the status byte, and loops until Toshiba GLCD module is ready.
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`T6963C_waitReady();`

T6963C_fill

Prototype	`void T6963C_fill(unsigned char v, unsigned int start, unsigned int len);`
Description	Fills controller memory block with given byte. Parameters : - `v`: byte to be written - `start`: starting address of the memory block - `len`: length of the memory block in bytes
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`T6963C_fill(0x33,0x00FF,0x000F);`

T6963C_dot

Prototype	`void T6963C_dot(int x, int y, unsigned char color);`
Description	Draws a dot in the current graphic panel of GLCD at coordinates (x, y). Parameters : - `x`: dot position on x-axis - `y`: dot position on y-axis - `color`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`T6963C_dot(x0, y0, pcolor);`

T6963C_write_char

Prototype	`void T6963C_write_char(unsigned char c, unsigned char x, unsigned char y, unsigned char mode);`
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	`void T6963C_write_text(unsigned char *str, unsigned char x, unsigned char y, unsigned char mode);`
Description	Writes text in the current text panel of GLCD at coordinates (x, y). Parameters : - `str`: text to be written - `x`: text position on x-axis - `y`: text position on y-axis - `mode`: mode parameter. Valid values: T6963C_ROM_MODE_OR, T6963C_ROM_MODE_XOR, T6963C_ROM_MODE_AND and T6963C_ROM_MODE_TEXT Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical “exclusive OR”. This can be useful to display text in the negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`T6963C_write_text(" GLCD LIBRARY DEMO, WELCOME !", 0, 0, T6963C_ROM_MODE_XOR);`

T6963C_line

Prototype	`void T6963C_line(int x0, int y0, int x1, int y1, unsigned char pcolor);`
Description	Draws a line from (x0, y0) to (x1, y1). Parameters : - `x0`: x coordinate of the line start - `y0`: y coordinate of the line end - `x1`: x coordinate of the line start - `y1`: y coordinate of the line end - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`T6963C_line(0, 0, 239, 127, T6963C_WHITE);`

T6963C_rectangle

Prototype	`void T6963C_rectangle(int x0, int y0, int x1, int y1, unsigned char pcolor);`
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	`void T6963C_box(int x0, int y0, int x1, int y1, unsigned char pcolor);`
Description	Draws a box on GLCD Parameters : - `x0`: x coordinate of the upper left box corner - `y0`: y coordinate of the upper left box corner - `x1`: x coordinate of the lower right box corner - `y1`: y coordinate of the lower right box corner - `pcolor`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`T6963C_box(0, 119, 239, 127, T6963C_WHITE);`

T6963C_circle

Prototype	`void T6963C_circle(int x, int y, long r, unsigned char pcolor);`
Description	Draws a circle on GLCD. Parameters : - `x`: x coordinate of the circle center - `y`: y coordinate of the circle center - `r`: radius size - `pcolor`: color parameter. Valid values: `T6963C_BLACK` and `T6963C_WHITE`
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`T6963C_circle(120, 64, 110, T6963C_WHITE);`

T6963C_image

Prototype	`void T6963C_image(const char *pic);`
Description	Displays bitmap on GLCD. Parameters : - `pic`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroC for dsPIC30 pointer to const and pointer to RAM equivalency). Use the mikroC'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(mc);`

T6963C_sprite

Prototype	`void T6963C_sprite(unsigned char px, unsigned char py, const char *pic, unsigned char sx, unsigned char sy);`
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	`void T6963C_set_cursor(unsigned char x, unsigned char y);`
Description	Sets cursor to row x and column y. Parameters : - `x`: cursor position row number - `y`: cursor position column number
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`T6963C_set_cursor(cposx, cposy);`

T6963C_clearBit

Prototype	`void T6963C_clearBit(unsigned int b);`
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	`void T6963C_setBit(unsigned int b);`
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	`void T6963C_negBit(unsigned int b);`
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	`void T6963C_displayGrPanel(unsigned int n);`
Description	Display selected graphic panel. Parameters : - `n`: graphic panel number. Valid values: 0 and 1.
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`// display graphic panel 1` `T6963C_displayGrPanel(1);`

T6963C_displayTxtPanel

Prototype	`void T6963C_displayTxtPanel(unsigned int n);`
Description	Display selected text panel. Parameters : - `n`: text panel number. Valid values: 0 and 1.
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`// display text panel 1` `T6963C_displayTxtPanel(1);`

T6963C_setGrPanel

Prototype	`void T6963C_setGrPanel(unsigned int n);`
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	`void T6963C_setTxtPanel(unsigned int n);`
Description	Compute start address for selected text panel and set appropriate internal pointers. All subsequent text operations will be preformed at this text panel. Parameters : - `n`: text panel number. Valid values: 0 and 1.
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`// set text panel 1 as current text panel.` `T6963C_setTxtPanel(1);`

T6963C_panelFill

Prototype	`void T6963C_panelFill(unsigned char v);`
Description	Fill current panel in full (graphic+text) with appropriate value (0 to clear). Parameters : - `v`: value to fill panel with.
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`// clear current panel` `T6963C_panelFill(0);`

T6963C_grFill

Prototype	`void T6963C_grFill(unsigned char v);`
Description	Fill current graphic panel with appropriate value (0 to clear). Parameters : - `v`: value to fill graphic panel with.
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`// clear current graphic panel` `T6963C_grFill(0);`

T6963C_txtFill

Prototype	`void T6963C_txtFill(unsigned char v);`
Description	Fill current text panel with appropriate value (0 to clear). Parameters : - `v`: this value increased by 32 will be used to fill text panel.
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`// clear current text panel` `T6963C_txtFill(0);`

T6963C_cursor_height

Prototype	`void T6963C_cursor_height(unsigned char n);`
Description	Set cursor size. Parameters : - `n`: cursor height. Valid values: `0..7`.
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`T6963C_cursor_height(7);`

T6963C_graphics

Prototype	`void T6963C_graphics(unsigned int n);`
Description	Enable/disable graphic displaying. Parameters : - `n`: on/off parameter. Valid values: 0 (disable graphic dispaying) and 1 (enable graphic displaying).
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`// enable graphic displaying` `T6963C_graphics(1);`

T6963C_text

Prototype	`void T6963C_text(unsigned int n);`
Description	Enable/disable text displaying. Parameters : - `n`: on/off parameter. Valid values: 0 (disable text dispaying) and 1 (enable text displaying).
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`// enable text displaying` `T6963C_text(1);`

T6963C_cursor

Prototype	`void T6963C_cursor(unsigned int n);`
Description	Set cursor on/off. Parameters : - `n`: on/off parameter. Valid values: 0 (set cursor off) and 1 (set cursor on).
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`// set cursor on` `T6963C_cursor(1);`

T6963C_cursor_blink

Prototype	`void T6963C_cursor_blink(unsigned int n);`
Description	Enable/disable cursor blinking. Parameters : - `n`: on/off parameter. Valid values: 0 (disable cursor blinking) and 1 (enable cursor blinking).
Requires	Toshiba GLCD module needs to be initialized. See the `T6963C_init` routine.
Example	`// enable cursor blinking` `T6963C_cursor_blink(1);`

Library Example

The following drawing demo tests advanced routines of the T6963C GLCD library. Hardware configurations in this example are made for the dsPICPRO2 board and dsPIC30F6014A.

```
/*
 * bitmap pictures stored in ROM
 */
extern const char mc[] ;
extern const char einstein[] ;

void main(void)
        {
        unsigned char   panel ;          // current panel
        unsigned int    i ;              // general purpose register
        unsigned char   curs ;           // cursor visibility
        unsigned int    cposx, cposy ;   // cursor x-y position

        ADPCFG = 0xFFFF;
        TRISD = 0xFFFF;
        TRISF = 0 ;                       // portF is output only
        PORTF = 0b00000000 ;             // chip enable, reverse on, 8x8 font


//continues...
```

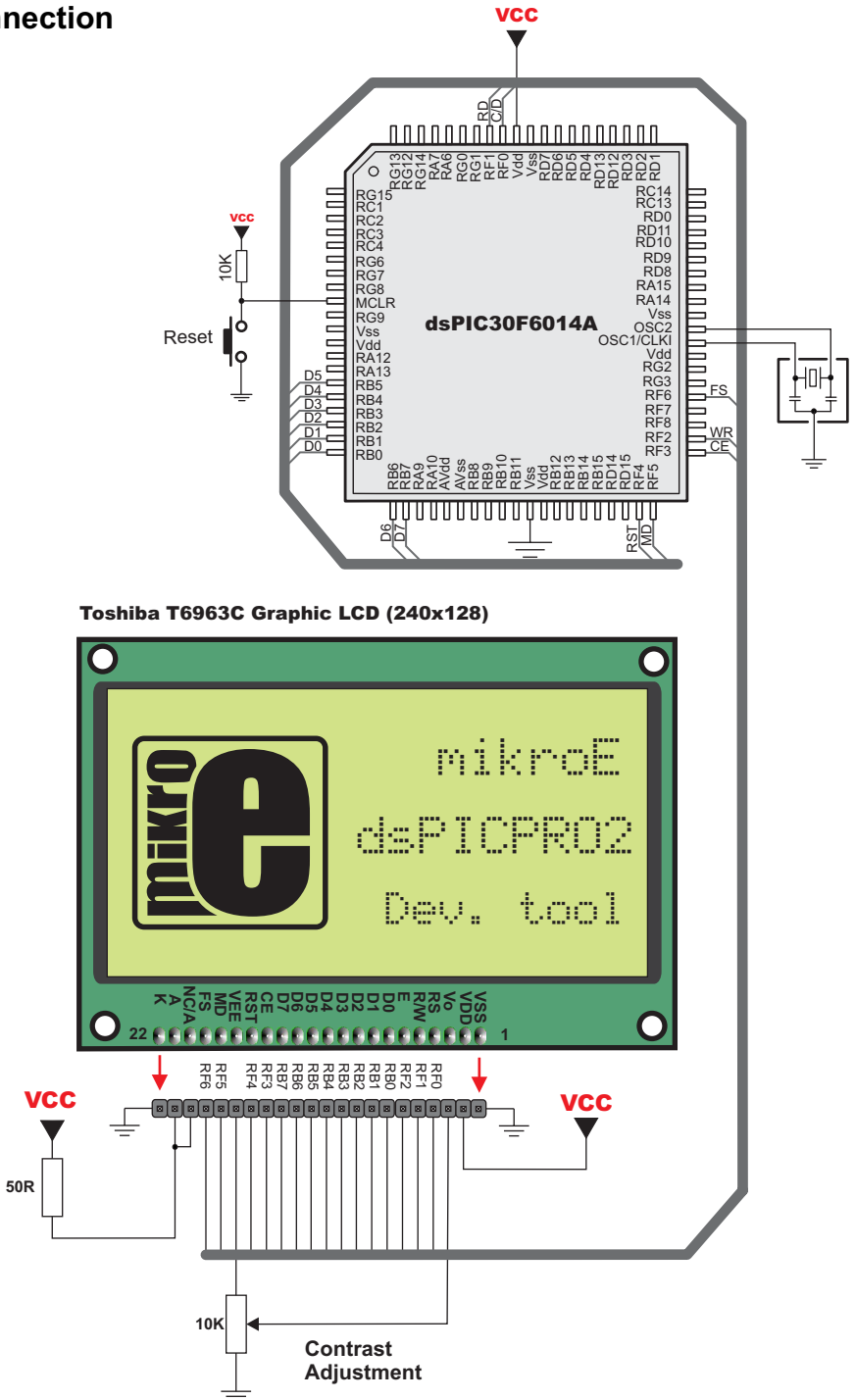
```
//continued...
        /*
          * init display for 240 pixel width and 128 pixel height
          * 8 bits character width
          * data bus on PORTB
          * control bus on PORTF
          * bit 2 is !WR
          * bit 1 is !RD
          * bit 0 is !CD
          * bit 4 is RST
          */
         T6963C_Init_240x128();
         //T6963C_init(240, 128, 8, &PORTB, &PORTF, 2, 1, 0, 4) ;
         /*
          * enable both graphics and text display at the same time
          */
         T6963C_graphics(1) ;
         T6963C_text(1) ;

         panel = 0 ;
         i = 0 ;
         curs = 0 ;
         cposx = cposy = 0 ;
         /*
          * text messages
          */
         T6963C_write_text(" GLCD LIBRARY DEMO, WELCOME !", 0, 0,
T6963C_ROM_MODE_XOR) ;
         T6963C_write_text(" EINSTEIN WOULD HAVE LIKED mC", 0, 15,
T6963C_ROM_MODE_XOR) ;

         /*
          * cursor
          */
         T6963C_cursor_height(8) ;        // 8 pixel height
         T6963C_set_cursor(0, 0) ;        // move cursor to top left
         T6963C_cursor(0) ;                // cursor off

         /*
          * draw rectangles
          */
         T6963C_rectangle(0, 0, 239, 127, T6963C_WHITE) ;
         T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE) ;
         T6963C_rectangle(40, 40, 199, 87, T6963C_WHITE) ;
         T6963C_rectangle(60, 60, 179, 67, T6963C_WHITE) ;



//continues...
```

```
//continued...

        /*
         * draw a cross
         */
        T6963C_line(0, 0, 239, 127, T6963C_WHITE) ;
        T6963C_line(0, 127, 239, 0, T6963C_WHITE) ;

        /*
         * draw solid boxes
         */
        T6963C_box(0, 0, 239, 8, T6963C_WHITE) ;
        T6963C_box(0, 119, 239, 127, T6963C_WHITE) ;

        /*
         * draw circles
         */
        T6963C_circle(120, 64, 10, T6963C_WHITE) ;
        T6963C_circle(120, 64, 30, T6963C_WHITE) ;
        T6963C_circle(120, 64, 50, T6963C_WHITE) ;
        T6963C_circle(120, 64, 70, T6963C_WHITE) ;
        T6963C_circle(120, 64, 90, T6963C_WHITE) ;
        T6963C_circle(120, 64, 110, T6963C_WHITE) ;
        T6963C_circle(120, 64, 130, T6963C_WHITE) ;

        T6963C_sprite(76, 4, einstein, 88, 119) ;
        // draw a sprite

        T6963C_setGrPanel(1) ;// select other graphic panel

        T6963C_image(mc) ;
        // fill the graphic screen with a picture

        for(;;)
                {
                /*
                 * if RD0 is pressed, toggle the display between
                 * graphic panel 0 and graphic 1
                 */
                if(PORTDbits.RD0)
                  {
                  panel++ ;
                  panel &= 1 ;
                  T6963C_displayGrPanel(panel) ;
                  Delay_ms(300) ;
                  }

//continues...
```

```
//continued...
                        /*
                         * if RD1 is pressed, display only graphic panel
                         */
                        else if(PORTDbits.RD1)
                                {
                                T6963C_graphics(1) ;
                                T6963C_text(0) ;
                                Delay_ms(300) ;
                                }

                        /*
                         * if RD2 is pressed, display only text panel
                         */
                        else if(PORTDbits.RD2)
                                {
                                T6963C_graphics(0) ;
                                T6963C_text(1) ;
                                Delay_ms(300) ;
                                }

                        /*
                         * if RD3 is pressed, display text and graphic
                         * panels
                         */
                        else if(PORTDbits.RD3)
                                {
                                T6963C_graphics(1) ;
                                T6963C_text(1) ;
                                Delay_ms(300) ;
                                }

                        /*
                         * if RD4 is pressed, change cursor
                         */
                        else if(PORTDbits.RD4)
                                {
                                curs++ ;
                                if(curs == 3) curs = 0 ;


//continues...
```

```
//continued...
                        switch(curs)
                                {
                                case 0:
                                        // no cursor
                                        T6963C_cursor(0) ;
                                        break ;
                                case 1:
                                        // blinking cursor
                                        T6963C_cursor(1) ;
                                        T6963C_cursor_blink(1) ;
                                        break ;
                                case 2:
                                        // non blinking cursor
                                        T6963C_cursor(1) ;
                                        T6963C_cursor_blink(0) ;
                                        break ;
                                }
                        Delay_ms(300) ;
                        }

                /*
                 * move cursor, even if not visible
                 */
                cposx++ ;
                if(cposx == T6963C_txtCols)
                        {
                        cposx = 0 ;
                        cposy++ ;
            if(cposy == T6963C_grHeight / T6963C_CHARACTER_HEIGHT);
                                {
                                cposy = 0 ;
                                }
                        }
                T6963C_set_cursor(cposx, cposy) ;

                Delay_ms(100) ;
                }
        } //~!
```

Hardware Connection



UART Library

The UART hardware module is available with a number of dsPIC30/33 and PIC24 MCUs. The mikroC 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           Uart2_Init           Uart_Init
Uart1_Init_Advanced  Uart2_Init_Advanced  Uart_Init_Advanced
Uart1_Data_Ready     Uart2_Data_Ready     Uart_Data_Ready
Uart1_Read_Char      Uart2_Read_Char      Uart_Read_Char
Uart1_Write_Char     Uart2_Write_Char     Uart_Write_Char
                                          Uart_Set_Active
```

Uart1_Init

Prototype	`void Uart1_Init(unsigned long baud_rate);`
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 for 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 `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` `Uart1_Init(2400);`

Uart1_Init_Advanced

Prototype (dsPIC30)	`void Uart1_Init_Advanced(unsigned long baud_rate, unsigned int parity, unsigned int stop_bits);`
Prototype (dsPIC33 and PIC24)	`void Uart1_Init_Advanced(unsigned long baud_rate, unsigned int parity, unsigned int stop_bits, unsigned int high_low_speed);`
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	`unsigned Uart1_Data_Ready(void);`
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 UART1 module. The UART1 module must be initialized before using this routine. See the Uart1_Init routine.

Example	`unsigned receive;`
	`...`
	`// read data if ready`
	`if (Uart1_Data_Ready())`
	`  receive = Uart1_Read_Char();`

Uart1_Read_Char

Prototype	`unsigned Uart1_Read_Char(void);`
Returns	Received byte.
Description	The function receives a byte via UART1. Use the `Uart1_Data_Ready` function to test if data is ready first.
Requires	MCU with the UART1 module. The UART1 module must be initialized before using this routine. See `Uart1_Init` routine.

Example	`unsigned receive;`
	`...`
	`// read data if ready`
	`if (Uart1_Data_Ready())`
	`  receive = Uart1_Read_Char();`

Uart2_Write_Char

Prototype	`void Uart1_Write_Char(unsigned char data);`
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 `Uart1_Init` routine.

Example	`unsigned char data = 0x1E;`
	`...`
	`Uart1_Write_Char(data);`

Uart2_Init

Prototype	`void Uart2_Init(unsigned long baud_rate);`
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 data sheet for baud rates allowed for specific Fosc. **Note**: For the dsPIC33 and pic24 MCUs, the compiler will choose for 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 `Uart2_Init_Advanced` routine, which allows speed selection, should be used.
Requires	MCU with the UART2 module.

Example	`// Initialize the UART2 module and establish communication at`
	`// 2400 bps`
	`Uart2_Init(2400);`

Uart2_Init_Advanced

Prototype (dsPIC30)	`void Uart2_Init_Advanced(unsigned long baud_rate, unsigned int parity, unsigned int stop_bits);`
Prototype (dsPIC33 and PIC24)	`void Uart2_Init_Advanced(unsigned long baud_rate, unsigned int parity, unsigned int stop_bits, unsigned int high_low_speed);`
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	`unsigned Uart2_Data_Ready(void);`
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	Routine requires the UART2 module. The UART2 module must be initialized before using this routine. See `Uart2_Init` routine.

Example	`unsigned receive;`
	`...`
	`// read data if ready`
	`if (Uart2_Data_Ready())`
	`  receive = Uart2_Read_Char();`

Uart2_Read_Char

Prototype	`unsigned Uart2_Read_Char(void);`
Returns	Received byte.
Description	The function receives a byte via UART2. Use the `Uart2_Data_Ready` function to test if data is ready first.
Requires	Routine requires the UART2 module. The UART2 module must be initialized before using this routine. See `Uart2_Init` routine.

Example	`unsigned receive;`
	`...`
	`// read data if ready`
	`if (Uart2_Data_Ready())`
	`  receive = Uart2_Read_Char();`

Uart2_Write_Char

Prototype	`void Uart2_Write_Char(unsigned char data);`
Description	The function transmits a byte via the UART2 module. Parameters : - `data`: data to be sent
Requires	Routine requires the UART2 module. The UART2 module must be initialized before using this routine. See `Uart2_Init` routine.

Example	`unsigned char data = 0x1E;`
	`...`
	`Uart2_Write_Char(data);`

Uart_Init

Prototype	`void Uart_Init(unsigned long baud_rate);`
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 for 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 `Uart_Init_Advanced` routine, which allows speed selection, should be used.
Requires	Routine requires the UART1 module.

Example	`// Initialize hardware UART and establish communication at`
	`// 2400 bps`
	`Uart_Init(2400);`

Uart_Init_Advanced

Prototype (dsPIC30)	`void Uart_Init_Advanced(unsigned long baud_rate, unsigned int parity, unsigned int stop_bits);`
Prototype (dsPIC33 and PIC24)	`void Uart_Init_Advanced(unsigned long baud_rate, unsigned int parity, unsigned int stop_bits, unsigned int high_low_speed);`
Description	Configures and initializes the UART 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	Routine requires 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	`unsigned Uart_Data_Ready(void);`
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 Uart_Set_Active routine.
Requires	Routine requires at least one UART module. Used UART module must be initialized before using this routine. See `Uart1_Init`, `Uart2_Init` and `Uart_Init` routines.

Example	`unsigned receive;`
	`...`
	`// read data if ready`
	`if (Uart_Data_Ready())`
	`  receive = Uart_Read_Char();`

Uart_Read_Char

Prototype	`unsigned Uart_Read_Char(void);`
Returns	Received byte.
Description	The function receives a byte via UART. Use the `Uart_Data_Ready` function to test if data is ready first. **Note**: if MCU has 2 UART modules, active module will be used. See `Uart_Set_Active` routine.
Requires	Routine requires at least one UART module. Used UART module must be initialized before using this routine. See `Uart1_Init`, `Uart2_Init` and `Uart_Init` routines.

Example	`unsigned receive;`
	`...`
	`// read data if ready`
	`if (Uart_Data_Ready())`
	`  receive = Uart_Read_Char();`

Uart_Write_Char

Prototype	`void Uart_Write_Char(unsigned char data);`
Description	The function transmits a byte via the UART module. Parameters : - `data`: data to be sent **Note**: if MCU has 2 UART modules, active module will be used. See Uart_Set_Active routine.
Requires	Routine requires at least one UART module. Used UART module must be initialized before using this routine. See `Uart1_Init`, `Uart2_Init` and `Uart_Init` routines.

Example	`unsigned char data = 0x1E;`
	`...`
	`Uart_Write_Char(data);`

Uart_Set_Active

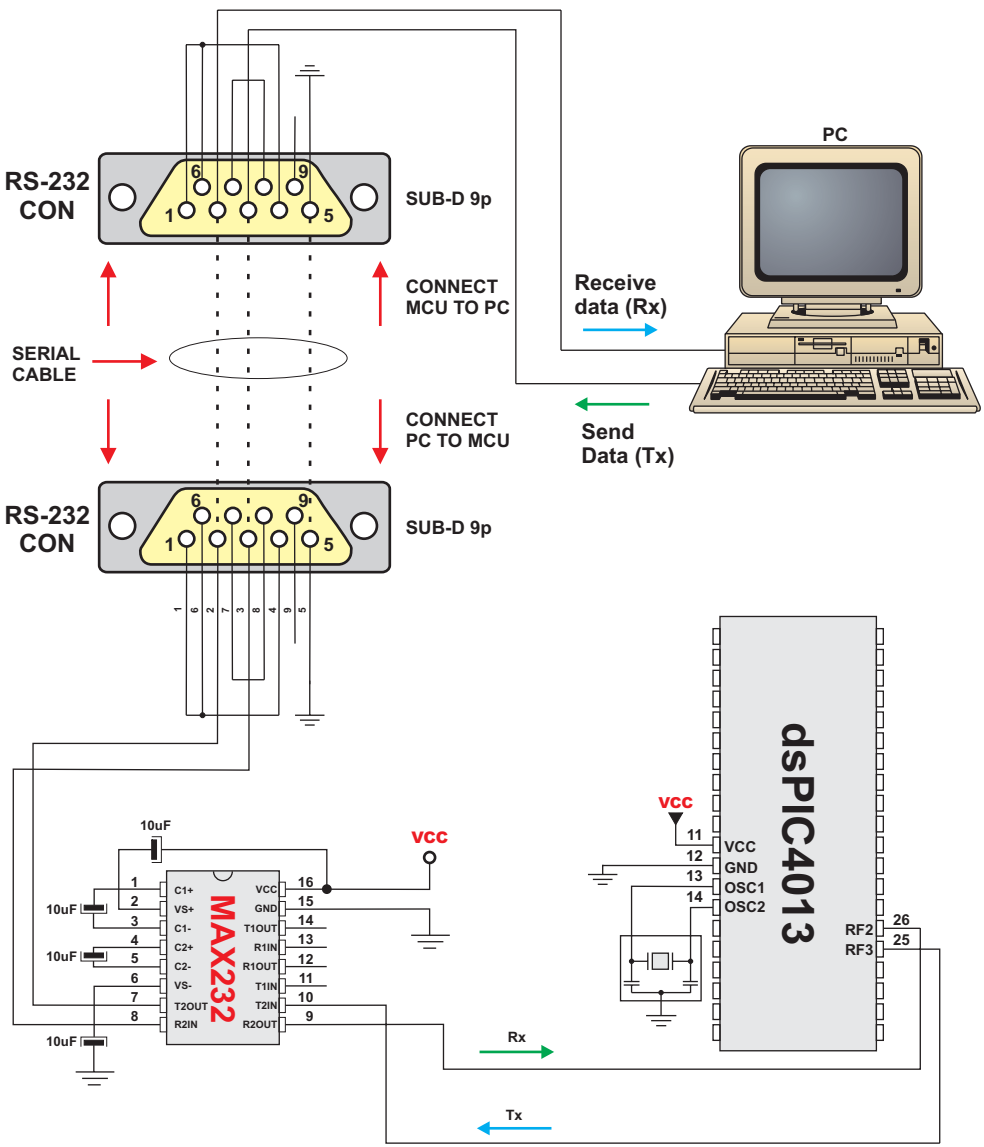
Prototype	`void Uart_Set_Active(char UartNo);`
Returns	Nothing.
Description	Sets active UART module which will be used by `Uart_Data_Ready`, `Uart_Read_Char` and `Uart_Write_Char` routines. Parameters : - `UartNo`: module number. Valid values: 1 (for UART1) and 2 (for UART2)
Requires	Routine is available only for MCUs with two UART modules. Used UART module must be initialized before using this routine. See `Uart1_Init`, `Uart2_Init` and `Uart_Init` routines.
Example	`// Activate UART2 module` `Uart_Set_Active(2);`

Uart_Set_Active

This example demonstrates simple data exchange via UART. If MCU is connected to the PC, you can test the example from the mikroC for dsPIC30/33 and PIC24 USART communication terminal, launch it from the drop-down menu **Tools › USART Terminal** or simply click the USART Terminal Icon.

```
unsigned rx1;

void main() {

  Uart1_Init(9600);
  Uart1_Write_Char('s');

  while(1)
  {
    if (Uart1_Data_Ready())  {
     rx1 = Uart1_Read_Char();
     Uart1_Write_Char(rx1);
    }
  }
} //~!
```

Hardware Connection



ANSI C Ctype Library

The mikroC for dsPIC30/33 and PIC24 provides a set of standard ANSI C library functions for testing and mapping characters.

Note: Not all of the standard functions have been included.

Note: The functions have been mostly implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate dsPIC30/33 and PIC24 programming. Be sure to skim through the description before using standard C functions.

Library Routines

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

isalnum

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

isalpha

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

iscntrl

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

isdigit

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

isgraph

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

islower

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

ispunct

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

isspace

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

isupper

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

isxdigit

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

toupper

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

tolower

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

ANSI C Math Library

The mikroC for dsPIC30/33 and PIC24 provides a set of standard ANSI C library functions for floating point math handling.

Note: Not all of the standard functions have been included.

Note: The functions have been mostly implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate dsPIC30/33 and PIC24 programming. Be sure to skim through the description before using standard C functions.

Library Routines

```
acos
asin
atan
atan2
ceil
cos
cosh
exp
fabs
floor
frexp
ldexp
log
log10
modf
pow
sin
sinh
sqrt
tan
tanh
```

acos

Prototype	`double acos(double x);`
Description	Function returns the arc cosine of parameter x; that is, the value whose cosine is x. The input parameter x must be between -1 and 1 (inclusive). The return value is in radians, between 0 and pi (inclusive).

asin

Prototype	`double asin(double x);`
Description	Function returns the arc sine of parameter x; that is, the value whose sine is x. 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	`double atan(double f);`
Description	Function computes the arc tangent of parameter f; that is, the value whose tangent is f. The return value is in radians, between -pi/2 and pi/2 (inclusive).

atan2

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

ceil

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

cos

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

cosh

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

exp

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

fabs

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

floor

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

frexp

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

ldexp

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

log

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

log10

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

modf

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

pow

Prototype	`double pow(double x, double y);`
Description	Function returns the value of `x` raised to the power of y (i.e. x^y). If the `x` is negative, function will automatically cast the y into `unsigned long`.

sin

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

sinh

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

sqrt

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

tan

Prototype	`double tan(double x);`
Description	Function returns the tangent of x in radians. The return value spans the allowed range of floating point in the mikroC for dsPIC30/33 and PIC24.

tanh

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

ANSI C Stdlib Library

The mikroC for dsPIC30/33 and PIC24 provides a set of standard ANSI C library functions of general utility.

Note: Not all of the standard functions have been included.

Note: Functions have been mostly implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate dsPIC30/33 and PIC24 programming. Be sure to skim through the description before using standard C functions.

Library Routines

```
abs
atof
atoi
atol
div
ldiv
labs
max
min
rand
srand
xtoi
```

abs

Prototype	`int abs(int a);`
Description	Function returns the absolute (i.e. positive) value of `a`.

atof

Prototype	`double atof(char *s)`
Description	Function converts the input string s into a double precision value and returns the value. Input string s should conform to the floating point literal format, with an optional white-space at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (including a null character).

atoi

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

atol

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

div

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

ldiv

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

labs

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

max

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

min

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

rand

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

srand

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

xtoi

Prototype	`unsigned xtoi(register char *s);`
Description	Function converts the input string `s` consisting of hexadecimal digits into an integer value. The input parameter `s` should consist exclusively of hexadecimal digits, with an optional whitespace and a sign at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (including a null character).

Div Structures

```
typedef struct divstruct {
            int quot;
            int rem;
} div_t;


typedef struct ldivstruct {
            long quot;
            long rem;
} ldiv_t;


typedef struct uldivstruct {
            unsigned long quot;
            unsigned long rem;
} uldiv_t;
```

ANSI C String Library

The mikroC for dsPIC30/33 and PIC24 provides a set of standard ANSI C library functions useful for manipulating strings and RAM memory.

Note: Not all of the standard functions have been included.

Note: Functions have been mostly implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate dsPIC30/33 and PIC24 programming. Be sure to skim through the description before using standard C functions.

Library Routines

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

memchr

Prototype	`void *memchr(void *p, unsigned int n, unsigned int v);`
Description	Function locates the first occurrence of int `n` in the initial `v` words of memory area starting at the address `p`. The function returns the pointer to this location or 0 if the `n` was not found. For parameter p you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example &mystring or &PORTB.

memcmp

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

memcpy

Prototype	`void *memcpy(void *d1, void *s2, int n);`
Description	Function copies `n` characters from the object pointed to by `s2` into the object pointed to by `d1`. If copying takes place between objects that overlap, the behavior is undefined. The function returns address of the object pointed to by d1.

memmove

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

memset

Prototype	`void *memset(void *p1, char character, int n);`
Description	Function copies the value of the `character` into each of the first n characters of the object pointed by `p1`. The function returns address of the object pointed to by `p1`.

strcat

Prototype	`char *strcat(char *to, char *from);`
Description	Function appends a copy of the string `from` to the string `to`, overwriting the null character at the end of `to`. Then, a terminating null character is added to the result. If copying takes place between objects that overlap, the behavior is undefined. to string must have enough space to store the result. The function returns address of the object pointed to by `to`.

strchr

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

strcmp

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

strcpy

Prototype	`char *strcpy(char *to, char *from);`
Description	Function copies the string from into the string to. If copying is successful, the function returns to. If copying takes place between objects that overlap, the behavior is undefined.

strlen

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

strncat

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

strncpy

Prototype	`char *strncpy(char *to, char *from, int size);`
Description	Function copies not more than size characters from string from to to. If copying takes place between objects that overlap, the behavior is undefined. If from is shorter than size characters, then to will be padded out with null characters to make up the difference. The function returns the resulting string to.

strspn

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

Strncmp

Prototype	`int strncmp(char *s1, char *s2, char len);`

Description	Function lexicographically compares not more than len characters (characters that follow the null character are not compared) from the string pointed by s1 to the string pointed by s2. The function returns a value indicating the s1 and s2 relationship:
	
	`Value  Meaning`
	`< 0  s1 "less than" s2`
	`= 0  s1 "equal to" s2`
	`> 0  s1 "greater than" s2`

Strstr

Prototype	`char *strstr(char *s1, char *s2);`
Description	Function locates the first occurrence of the string s2 in the string s1 (excluding the terminating null character). The function returns pointer to first occurrence of s2 in s1; if no string was found, function returns 0. If s2 is a null string, the function returns 0.

Strcspn

Prototype	`char *strcspn(char * s1, char *s2);`
Description	Function computes the length of the maximum initial segment of the string pointed to by s1 that consists entirely of characters that are not in the string pointed to by s2. The function returns the length of the initial segment.

Strpbrk

Prototype	`char *strpbrk(char * s1, char *s2);`
Description	Function searches s1 for the first occurrence of any character from the string s2. The terminating null character is not included in the search. The function returns pointer to the matching character in s1. If s1 contains no characters from s2, the function returns 0.

Strrchr

Prototype	`char *strrchr(char * ptr, unsigned int chr);`
Description	Function searches the string ptr for the last occurrence of character chr. The null character terminating ptr is not included in the search. The function returns pointer to the last chr found in ptr; if no matching character was found, function returns 0.

Conversions Library

The mikroC 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
IntToStr
LongToStr
LongWordToStr
FloatToStr
```

The following functions convert decimal values to BCD and vice versa:

```
Dec2Bcd
Bcd2Dec16
Dec2Bcd16
```

ByteToStr

Prototype	`void ByteToStr(unsigned short input, char *output);`
Description	Converts input byte to a string. The output string has fixed width of 4 characters including null character at the end (string termination). The output string is right justified and remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: byte to be converted - `output`: destination string
Requires	Destination string should be at least 4 characters in length.

Example	`unsigned short t = 24;`
	`char txt[4];`
	`...`
	`ByteToStr(t, txt);  // txt is " 24" (one blank here)`

ShortToStr

Prototype	`void ShortToStr(short input, char *output);`
Description	Converts input signed short number to a string. The output string has fixed width of 5 characters including null character at the end (string termination). The output string is right justified and remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: signed short number to be converted - `output`: destination string
Requires	Destination string should be at least 5 characters in length.

Example	`short t = -24;`
	`char txt[5];`
	`...`
	`ShortToStr(t, txt);  // txt is " -24" (one blank here)`

WordToStr

Prototype	`void WordToStr(unsigned input, char *output);`
Description	Converts input word to a string. The output string has fixed width of 6 characters including null character at the end (string termination). The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: word to be converted - `output`: destination string
Requires	Destination string should be at least 6 characters in length.

Example	`unsigned t = 437;`
	`char txt[6];`
	`...`
	`WordToStr(t, txt);  // txt is "  437" (two blanks here)`

IntToStr

Prototype	`void IntToStr(int input, char *output);`
Description	Converts input signed integer number to a string. The output string has fixed width of 7 characters including null character at the end (string termination). The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: signed integer number to be converted - `output`: destination string
Requires	Destination string should be at least 7 characters in length.

Example	`int j = -4220;`
	`char txt[7];`
	`...`
	`IntToStr(j, txt);  // txt is " -4220" (one blank here)`

LongToStr

Prototype	`void LongToStr(long input, char *output);`
Description	Converts input signed long integer number to a string. The output string has fixed width of 12 characters including null character at the end (string termination). The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: signed long integer number to be converted - `output`: destination string
Requires	Destination string should be at least 12 characters in length.

Example	`long jj = -3700000;`
	`char txt[12];`
	`...`
	`LongToStr(jj, txt);`
	`// txt is "   -3700000" (three blanks here)`

LongWordToStr

Prototype	`void LongWordToStr(unsigned long input, char *output);`
Description	Converts input unsigned long integer number to a string. The output string has fixed width of 11 characters including null character at the end (string termination). The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: unsigned long integer number to be converted - `output`: destination string
Requires	Destination string should be at least 11 characters in length.

Example	`unsigned long jj = 3700000;`
	`char txt[11];`
	`...`
	`LongToStr(jj, txt);`
	`// txt is "   3700000" (three blanks here)`

FloatToStr

Prototype	`unsigned char FloatToStr(float fnum, unsigned char *str);`
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.
Requires	Destination string should be at least 14 characters in length.

Example	`float ff1 = -374.2;`
	`float ff2 = 123.456789;`
	`float ff3 = 0.000001234;`
	`char txt[15];`
	`...`
	`FloatToStr(ff1, txt);  // txt is "-374.2"`
	`FloatToStr(ff2, txt);  // txt is "123.4567"`
	`FloatToStr(ff3, txt);  // txt is "1.234e-6"`

Dec2Bcd

Prototype	`unsigned short Dec2Bcd(unsigned short decnum);`
Returns	Converted BCD value.
Description	Converts input unsigned short integer number to its appropriate BCD representation. Parameters : - `decnum`: unsigned short integer number to be converted

Example	`unsigned short a, b;`
	`...`
	`a = 22;`
	`b = Dec2Bcd(a);  // b equals 34`

Bcd2Dec16

Prototype	`unsigned Bcd2Dec16(unsigned bcdnum);`
Returns	Nothing.
Description	Converts 16-bit BCD numeral to its decimal equivalent. Parameters : - `bcdnum`: 16-bit BCD numeral to be converted

Example	`unsigned a, b;`
	`...`
	`a = 0x1234;  // a equals 4660`
	`b = Bcd2Dec16(a);  // b equals 1234`

Dec2Bcd16

Prototype	`unsigned Dec2Bcd16(unsigned decnum);`
Returns	Converted BCD value.
Description	Converts unsigned 16-bit decimal value to its BCD equivalent. Parameters : - `decnum`: unsigned 16-bit decimal number to be converted

Example	`unsigned a, b;`
	`...`
	`a = 2345;`
	`b = Dec2Bcd16(a);  // b equals 9029`

Setjmp Library

The Setjmp library contains functions and types definitions for bypassing the normal function call and return discipline.

jmp_buf is an array of unsigned int type suitable for holding information needed to restore a calling environment. Type declaration is contained in the sejmp.h header file which can be found in the include folder of the compiler.

Library Routines

```
Setjmp
Longjmp
```

Setjmp

Prototype	`int Setjmp(jmp_buf env);`
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
Requires	Nothing.

Example	`jmp_buf buf;`
	`...`
	`Setjmp(buf);`

Longjmp

Prototype	`void Longjmp(jmp_buf env, int val);`
Returns	Nothing.
Description	Restores calling environment saved in 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	`jmp_buf buf;`
	`...`
	`Longjmp(buf, 2);`

Library Example

This example demonstrates function cross calling using the Setjmp and Longjmp functions. When called, Setjmp saves its calling environment in its **jmp_buf** argument for a later use by Longjmp. Longjmp, on the other hand, restores the environment saved by the most recent invocation of Setjmp with the corresponding jmp_buf argument.

```
#include <Setjmp.h>

jmp_buf buf;  // Note: Program flow diagrams are indexed according
              //       to the sequence of execution

void func33(){

  Delay_ms(1000);

  asm nop;
  Longjmp(buf, 2);
  asm nop;

}

void func(){

  portb = 3;
  if (Setjmp(buf) == 2)
    portb = 1;
  else
    func33();


}

void main() {

  PORTB = 0;
  TRISB = 0;

  asm nop;

  func();

  asm nop;
  Delay_ms(1000);
  PORTB = 0xFFFF;
}
```

Sprint Library

The mikroC for dsPIC30/33 and PIC24 provides the standard ANSI C Sprintf function for easy data formatting.

Note: In addition to ANSI C standard, the Sprint Library also includes two limited versions of the sprintf function (`sprinti` and `sprintl`). These functions take less ROM and RAM and may be more convenient for use in some cases.

Library Routines

```
sprintf
sprintl
sprinti
```

sprintf

Prototype	`sprintf(char *wh, const char *f,...);`
Returns	The function returns the number of characters actually written to destination string.
Description	sprintf is used to format data and print them into destination string. Parameters : - `wh`: destination string - `f`: format string The `f` argument is a format string and may be composed of characters, escape sequences, and format specifications. Ordinary characters and escape sequences are copied to the destination string in the order in which they are interpreted. Format specifications always begin with a percent sign (%) and require additional arguments to be included in the function call. The format string is read from left to right. The first format specification encountered refers to the first argument after f and then converts and outputs it using the format specification. The second format specification accesses the second argument after f, and so on. If there are more arguments than format specifications, then these extra arguments are ignored. Results are unpredictable if there are not enough arguments for the format specifications. The format specifications have the following format: `% [ flags] [ width] [ .precision] [{ l | L }] conversion_type` Each field in the format specification can be a single character or a number which specifies a particular format option. The `conversion_type` field is where a single character specifies that the argument is interpreted as a character, string, number, or pointer, as shown in the following table: *// continues on the next page ...*

Description

// continued from previous page ...

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

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

// continues on the next page ...

Description

// continued from previous page ...

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

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

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

// continues on the next page ...

Description

// continued from previous page ...

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

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

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

sprintl

Prototype	`sprintl(char *wh, const char *f,...);`
Returns	The function returns the number of characters actually written to destination string.
Description	The same as sprintf, except it doesn't support float-type numbers.

sprinti

Prototype	`sprinti(char *wh, const char *f,...);`
Returns	The function returns the number of characters actually written to destination string.
Description	The same as sprintf, except it doesn't support long integers and float-type numbers.

Library Example

This is a demonstration of usage of some standard C library functions located in the `c_math` library. Various calculations are performed and the result is being printed on Lcd (8bit mode) using the `Sprintf` function.

```
#define PI 3.14159265358979
double ww, pp, xx = 1.2587538e+1;
char  niz[30];

void main(){
  ADPCFG = 0xFFFF;

  Lcd8_Custom_Config(&PORTD,7,6,5,4,3,2,1,0, &PORTB, 4, 5, 6);
  Lcd8_Custom_Cmd(LCD_CLEAR);
  Lcd8_Custom_Cmd(LCD_CURSOR_OFF);

  while(1){
          Lcd8_Custom_Out(1,1,"    Easy_math   ");
          Lcd8_Custom_Out(2,1,"       demo:    ");
          Delay_ms(2000);
          Lcd8_Custom_Cmd(LCD_CLEAR);

          pp = sin(PI/2.);
          sprintf(niz, "   sin(PI/2)=");

// continues ...
```

```
// continued ...

        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%9g", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);

        pp = cos(PI);
        sprintf(niz, "    cos(PI)=" );
        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%9g", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);

        pp = sinh(PI/4.);
        sprintf(niz, "   sinh(PI/4)=");
        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%12.5g", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);

        pp = cosh(PI/4.);
        sprintf(niz, "   cosh(PI/4)=");
        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%11.5g", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);

        pp = tan(PI/4.);
        sprintf(niz, "   tan(PI/4)=");
        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%8g", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);

        pp = asin(sin(PI/2.));
        sprintf(niz, "asin(sin(PI/2))=");
        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%11.5g", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);

// continues ...
```

```
// continued ...

        pp = acos(cos(PI));
        sprintf(niz, " acos(cos(PI))=");
        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%12g", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);

        pp = asin(sin(PI/4.));
        sprintf(niz, "asin(sin(PI/4))=");
        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%12.5g", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);

        pp = acos(cos(PI/4.));
        sprintf(niz, "acos(cos(PI/4))=");
        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%12.5g", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);

        pp = atan(tan(PI/4.));
        sprintf(niz, "atan(tan(PI/4))=");
        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%12.5g", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);

        pp = exp(xx);
        sprintf(niz, " exp(%g) =", xx);
        Lcd8_Custom_Out(1,1,niz);
        sprintf(niz, "%14e", pp);
        Lcd8_Custom_Out(2,1,niz);
        Delay_ms(2000);
        Lcd8_Custom_Cmd(LCD_CLEAR);


// continues ...
```

```
// continued ...

            ww = log10(xx);
            sprintf(niz, "log10(%g)=", xx);
            Lcd8_Custom_Out(1,1,niz);
            sprintf(niz, "%12g", ww);
            Lcd8_Custom_Out(2,1,niz);
            Delay_ms(2000);
            Lcd8_Custom_Cmd(LCD_CLEAR);

            ww = pow(xx,2.);
            sprintf(niz, " pow(%g,%g)=", xx, 2.);
            Lcd8_Custom_Out(1,1,niz);
            sprintf(niz, "%12g", ww);
            Lcd8_Custom_Out(2,1,niz);
            Delay_ms(2000);
            Lcd8_Custom_Cmd(LCD_CLEAR);

            pp =
pow(atan(exp(log(sin(cos(PI/4.))))),atan(exp(log(cos(sin(PI/4.)))))));
            sprintf(niz, "  complex_exp=");
            Lcd8_Custom_Out(1,1,niz);
            sprintf(niz, "%12g", pp);
            Lcd8_Custom_Out(2,1,niz);
            Delay_ms(2000);
            Lcd8_Custom_Cmd(LCD_CLEAR);

  }
} //~!
```

Time Library

The Time Library contains functions and type definitions for time calculations in the UNIX time format which counts the number of seconds since the "epoch". This is very convenient for programs that work with time intervals: the difference between two UNIX time values is a real-time difference measured in seconds.

What is the epoch?
Originally it was defined as the beginning of 1970 GMT. (January 1, 1970 Julian day) GMT, Greenwich Mean Time, is a traditional term for the time zone in England.

The TimeStruct type is a structure type suitable for time and date storage. Type declaration is contained in `timelib.h` which can be found in the mikroC for dsPIC30/33 and PIC24 Time Library Demo example folder.

Library Routines

```
Time_dateToEpoch
Time_epochToDate
Time_dateDiff
```

Time_dateToEpoch

Prototype	`long Time_dateToEpoch(TimeStruct *ts);`
Returns	Number of seconds since January 1, 1970 0h00mn00s.
Description	This function returns the unix time : number of seconds since January 1, 1970 0h00mn00s. Parameters : - `ts`: time and date value for calculating unix time.
Requires	Nothing.

Example	`#include "timelib.h"`
	`...`
	`TimeStruct ts1;`
	`long epoch ;`
	`...`
	`/*`
	`* what is the epoch of the date in ts ?`
	`*/`
	`epoch = Time_dateToEpoch(&ts1) ;`

Time_epochToDate

Prototype	`void Time_epochToDate(long e, TimeStruct *ts);`
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	`#include "timelib.h"`
	`...`
	`TimeStruct ts2;`
	`long epoch ;`
	`...`
	`/*`
	`* what date is epoch 1234567890 ?`
	`*/`
	`epoch = 1234567890 ;`
	`Time_epochToDate(epoch, &ts2) ;`

Time_dateDiff

Prototype	`long Time_dateDiff(TimeStruct *t1, TimeStruct *t2);`
Returns	Time difference in seconds as a signed long.
Description	This function compares two dates and returns time difference in seconds as a signed long. Result is positive if t1 is before t2, result is null if t1 is the same as t2 and result is negative if t1 is after t2. Parameters : - `t1`: time and date structure (the first comparison parameter) - `t2`: time and date structure (the second comparison parameter) **Note**: This function is implemented as macro in the `timelib.h` header file which can be found in the mikroC for dsPIC30/33 and PIC24 Time Library Demo example folder.

Example	`#include "timelib.h"`
	`...`
	`TimeStruct ts1, ts2;`
	`long diff ;`
	`...`
	`/*`
	`* how many seconds between these two dates contained in ts1 and`
	`* ts2 buffers?`
	`*/`
	`diff = Time_dateDiff(&ts1, &ts2) ;`

Library Example

This example demonstrates TimeLibraryDemo (simplified c-like time library for the dsPIC MCU).

```
#include         "timelib.h"

char     *libday[ ]  =
         {
         "monday",
         "tuesday",
         "wednesday",
         "thursday",
         "friday",
         "saturday",
         "sunday"
         } ;
TimeStruct ts1, ts2 ;

main()
         {
         char     buf[ 256]  ;
         long     epoch ;
         long     diff ;

         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) ;

         *buf = 0 ;
         sprintf(  buf, (const char *)"epoch of %.2d/%.2d/%.4d
%.2dh%.2dmn%.2ds (GMT+0) is : %ld, it was a %s\n",
                     (unsigned) ts1.md,
                     (unsigned) ts1.mo,
                     (unsigned) ts1.yy,
                     (unsigned) ts1.hh,
                     (unsigned) ts1.mn,
                     (unsigned) ts1.ss,
                     epoch,
                     libday[ ts1.wd]
                   ) ;

// continues ...
```

```
// continued ...

        /*
         * what date is epoch 1234567890 ?
         */


        epoch = 1234567890 ;
        Time_epochToDate(epoch, &ts2) ;
        sprintf(buf, (const char *)"Please, celebrate epoch %ld on
%.2d/%.2d/%.4d at %.2dh%.2dmn%.2ds (GMT+0), it will be a %s\n",
                        epoch,
                        (unsigned) ts2.md,
                        (unsigned) ts2.mo,
                        (unsigned) ts2.yy,
                        (unsigned) ts2.hh,
                        (unsigned) ts2.mn,
                        (unsigned) ts2.ss,
                        libday[ ts2.wd]
                        ) ;



        /*
         * how many seconds between these two dates there are ?
         */


        diff = Time_dateDiff(&ts1, &ts2) ;
        sprintf(buf, (const char *)"There are %ld seconds from
%.2d/%.2d/%.4d %.2dh%.2dmn%.2ds to %.2d/%.2d/%.4d
%.2dh%.2dmn%.2ds\n",
                        diff,
                        (unsigned) ts1.md,
                        (unsigned) ts1.mo,
                        (unsigned) ts1.yy,
                        (unsigned) ts1.hh,
                        (unsigned) ts1.mn,
                        (unsigned) ts1.ss,
                        (unsigned) ts2.md,
                        (unsigned) ts2.mo,
                        (unsigned) ts2.yy,
                        (unsigned) ts2.hh,
                        (unsigned) ts2.mn,
                        (unsigned) ts2.ss
                        ) ;
        } //~!
```

Trigonometry Library

The mikroC for dsPIC30/33 and PIC24 implements fundamental trigonometry functions. These functions are implemented as look-up tables. Trigonometry functions are implemented in integer format in order to save memory.

Library Routines

```
SinE3
CosE3
```

SinE3

Prototype	`int sinE3(unsigned angle_deg);`
Returns	The function returns the sine of input parameter.
Description	The function calculates sine multiplied by 1000 and rounded up to the nearest integer: `result = round_up(sin(angle_deg)*1000)` Parameters : - `angle_deg`: input angle in degrees **Note**: Return value range: `-1000..1000`.

Example	`int res;`
	`...`
	`res = sinE3(45);  // result is 707`

CosE3

Prototype	`int cosE3(unsigned angle_deg);`
Returns	The function returns the cosine of input parameter.
Description	The function calculates cosine multiplied by 1000 and rounded up to the nearest integer: `result = round_up(cos(angle_deg)*1000)` Parameters : - `angle_deg`: input angle in degrees **Note**: Return value range: -1000..1000.

Example	`int res;`
	`...`
	`res = cosE3(196);  // result is -193`

Util Library

The Util library contains miscellaneous routines useful for a project development.

Button

Prototype	`unsigned int Button(unsigned int *port, unsigned int pin, unsigned int time, unsigned int active_state);`
Returns	- `255` 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 then again after the debouncing period has expired. If the pin was in the active state in both cases then the function returns 255 (true). Parameters : - `port`: button port address - `pin`: button pin - `time`: debouncing period in milliseconds - `active_state`: determines what is considered as active state. Valid values: 0 (logical zero) and 1 (logical one)

Example	Read RB0, to which the button is connected to and invert PORTD on transition from "1" to "0" (release the button):
	
	`unsigned int oldstate;`
	
	`void main() {`
	`  ADPCFG = 0xFFFF;`
	`  TRISB = 0xFFFF;`
	`  TRISD = 0x0000;`
	
	`  do {`
	`  if (Button(&PORTB, 0, 1, 1))`
	`  oldstate = 1;`
	`  if (oldstate && Button(&PORTB, 0, 1, 0)) {`
	`  LATD = ~LATD;`
	`  oldstate = 0;`
	`  }`
	`  } while(1);`
	`}`

PrintOut

Prototype	`void PrintOut(void (*prntoutfunc)(char ch), const char *f,...);`
Returns	Nothing.

Description	PrintOut is used to format data and print them in a way defined by the user through a print handler function.
	
	Parameters :
	
	- `prntoutfunc`: print handler function
	- `f`: format string
	
	The f argument is a format string and may be composed of characters, escape sequences, and format specifications. Ordinary characters and escape sequences are copied to the print handler in order in which they are interpreted. Format specifications always begin with a percent sign (%) and require additional arguments to be included in the function call.
	
	The format string is read from left to right. The first format specification encountered refers to the first argument after the f parameter and then converts and outputs it using the format specification. The second format specification accesses the second argument after f, and so on. If there are more arguments than format specifications, the extra arguments are ignored. Results are unpredictable if there are not enough arguments for the format specifications. The format specifications have the following format:
	
	`% [ flags] [ width] [ .precision] [{ l | L }] conversion_type`
	
	Each field in the format specification can be a single character or a number which specifies a particular format option. The *conversion_type* field is where a single character specifies that an argument is interpreted as a character, string, number, or pointer, as shown in the following table:
	
	`// continues ...`

Description

// continued ...

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

// continues ...

Description

// continued ...

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

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

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

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

// continues ...

Description

// continued ...

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

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

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

PrintOut Example

```
void PrintHandler(char c){

  Uart1_Write_Char(c);

}

void main(){
  Uart1_Init(9600);
  Delay_ms(100);

  PrintOut(PrintHandler, "/*\r\n"
                         " * Project name:\r\n"
                         "     PrintOutExample (Sample usage of
PrintOut() function)\r\n"
                         " * Copyright:\r\n"
                         "     (c) MikroElektronika, 2006.\r\n"
                         " * Revision History:\r\n"
                         "     20060710:\r\n"
                         "       - Initial release\r\n"
                         " * Description:\r\n"
                         "     Simple demonstration on usage of the
PrintOut() function\r\n"
                         " * Test configuration:\r\n"
                         "     MCU:              dsPIC30F4013\r\n"
                         "     Dev.Board:        EASYdsPIC2\r\n"
                         "     Oscillator:       EC, %6.3fMHz\r\n"
                         "     Ext. Modules:     None.\r\n"
                         "     SW:               mikroC for dsPIC30/33
and PIC24 v3.0.0.0.\r\n"
                         " * NOTES:\r\n"
                         "     None.\r\n"
                         " */\r\n", Get_Fosc_kHz()/1000.);

} //~!
```

Contact us:

If you are experiencing problems with any of our products or you just want additional information, please let us know.

Technical Support for compiler

If you are experiencing any trouble with mikroC for dsPIC30/33 and PIC24, please do not hesitate to contact us - it is in our mutual interest to solve these issues.

Discount for schools and universities

mikroElektronika offers a special discount for educational institutions. If you would like to purchase mikroC (dsPIC30/33 and PIC24) for purely educational purposes, please contact us.

Problems with transport or delivery

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

Would you like to become mikroElektronika's distributor?

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

Other

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

mikroElektronika
Admirala Geprata 1B
11000 Belgrade
EUROPE

Phone: + 381 (11) 30 66 377, + 381 (11) 30 66 378
Fax: + 381 (11) 30 66 379
E-mail: office@mikroe.com
Website: http://www.mikroe.com/
Forums: http://www.mikroe.com/forum/
Support: http://www.mikroe.com/en/support/