# Table of Contents

Overview .................................................................................................................. 15  
C Compiler .................................................................................................................. 15  
PCD ............................................................................................................................... 15  
Technical Support ....................................................................................................... 16  
Directories .................................................................................................................... 16  
File Formats .................................................................................................................. 17  
Invoking the Command Line Compiler ....................................................................... 19  
PCW Overview ............................................................................................................. 21  
Menu ............................................................................................................................... 22  
Editor Tabs .................................................................................................................... 22  
Slide Out Windows ....................................................................................................... 22  
Editor ............................................................................................................................. 23  
Debugging Windows ..................................................................................................... 23  
Status Bar ....................................................................................................................... 24  
Output Messages .......................................................................................................... 24  
Program Syntax ............................................................................................................ 25  
Overall Structure ......................................................................................................... 25  
Comment ....................................................................................................................... 25  
Trigraph Sequences ..................................................................................................... 27  
Multiple Project Files .................................................................................................. 27  
Multiple Compilation Units .......................................................................................... 28  
Full Example Program ................................................................................................. 28  
Statements ..................................................................................................................... 31  
Statements ..................................................................................................................... 31  
if ..................................................................................................................................... 32  
while ............................................................................................................................... 33  
do-while ......................................................................................................................... 33  
for ................................................................................................................................... 33  
switch ............................................................................................................................. 34  
return .............................................................................................................................. 35  
goto ................................................................................................................................. 35  
label ............................................................................................................................... 35  
break ............................................................................................................................... 36  
continue .......................................................................................................................... 36  
expr ................................................................................................................................. 36  
; ................................................................................................................................... 37  
stmt ............................................................................................................................... 37  
Expressions ..................................................................................................................... 39  
Constants ....................................................................................................................... 39  
Identifiers ....................................................................................................................... 40  
Operators ....................................................................................................................... 40
## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator Precedence</td>
<td>42</td>
</tr>
<tr>
<td>Data Definitions</td>
<td>43</td>
</tr>
<tr>
<td>Data Definitions</td>
<td>43</td>
</tr>
<tr>
<td>Type Specifiers</td>
<td>43</td>
</tr>
<tr>
<td>Type Qualifiers</td>
<td>44</td>
</tr>
<tr>
<td>Enumerated Types</td>
<td>46</td>
</tr>
<tr>
<td>Structures and Unions</td>
<td>46</td>
</tr>
<tr>
<td>typedef</td>
<td>47</td>
</tr>
<tr>
<td>Non-RAM Data Definitions</td>
<td>48</td>
</tr>
<tr>
<td>Using Program Memory for Data</td>
<td>50</td>
</tr>
<tr>
<td>Named Registers</td>
<td>52</td>
</tr>
<tr>
<td>Function Definition</td>
<td>53</td>
</tr>
<tr>
<td>Function Definition</td>
<td>53</td>
</tr>
<tr>
<td>Overloaded Functions</td>
<td>54</td>
</tr>
<tr>
<td>Reference Parameters</td>
<td>54</td>
</tr>
<tr>
<td>Default Parameters</td>
<td>55</td>
</tr>
<tr>
<td>Variable Argument Lists</td>
<td>55</td>
</tr>
<tr>
<td>Functional Overview</td>
<td>57</td>
</tr>
<tr>
<td>I2C</td>
<td>57</td>
</tr>
<tr>
<td>ADC</td>
<td>58</td>
</tr>
<tr>
<td>Analog Comparator</td>
<td>59</td>
</tr>
<tr>
<td>CAN Bus</td>
<td>60</td>
</tr>
<tr>
<td>Code Profile</td>
<td>64</td>
</tr>
<tr>
<td>Configuration Memory</td>
<td>65</td>
</tr>
<tr>
<td>CRC</td>
<td>66</td>
</tr>
<tr>
<td>DAC</td>
<td>67</td>
</tr>
<tr>
<td>Data Eeprom</td>
<td>68</td>
</tr>
<tr>
<td>DCI</td>
<td>69</td>
</tr>
<tr>
<td>DMA</td>
<td>70</td>
</tr>
<tr>
<td>Data Signal Modulator</td>
<td>72</td>
</tr>
<tr>
<td>Extended RAM</td>
<td>73</td>
</tr>
<tr>
<td>General Purpose I/O</td>
<td>74</td>
</tr>
<tr>
<td>Input Capture</td>
<td>75</td>
</tr>
<tr>
<td>Internal Oscillator</td>
<td>76</td>
</tr>
<tr>
<td>Interrupts</td>
<td>77</td>
</tr>
<tr>
<td>Output Compare/PWM Overview</td>
<td>78</td>
</tr>
<tr>
<td>Motor Control PWM</td>
<td>79</td>
</tr>
<tr>
<td>PMP/EPMP</td>
<td>81</td>
</tr>
<tr>
<td>Program Eeprom</td>
<td>82</td>
</tr>
<tr>
<td>QEI</td>
<td>84</td>
</tr>
<tr>
<td>RS232 I/O</td>
<td>85</td>
</tr>
<tr>
<td>RTCC</td>
<td>88</td>
</tr>
<tr>
<td>RTOS</td>
<td>89</td>
</tr>
<tr>
<td>SPI</td>
<td>91</td>
</tr>
<tr>
<td>Timers</td>
<td>93</td>
</tr>
</tbody>
</table>
TimerA.................................................................................................................94
TimerB......................................................................................................................95
Voltage Reference.................................................................................................96
WDT or Watch Dog Timer.....................................................................................97
interrupt_enabled()............................................................................................98
Stream I/O..............................................................................................................98
PreProcessor .........................................................................................................101
PRE-PROCESSOR DIRECTORY ...........................................................................101
__address__...........................................................................................................101
__attribute__.........................................................................................................102
#asm  #endasm  #asm asis ..................................................................................103
#bank_dma...........................................................................................................111
#bankx................................................................................................................112
#banky................................................................................................................112
#bit........................................................................................................................113
__buildcount__.....................................................................................................114
#build...................................................................................................................114
#byte.....................................................................................................................116
#case......................................................................................................................116
__date__................................................................................................................117
#define..................................................................................................................118
definedinc............................................................................................................119
#device..................................................................................................................120
__device__..............................................................................................................123
#if expr  #else  #elif  #endif...............................................................................124
#error...................................................................................................................125
#export (options)...............................................................................................125
__file__..................................................................................................................127
__filename__.........................................................................................................127
#fill_rom...............................................................................................................128
#fuses....................................................................................................................128
#hexcomment.....................................................................................................129
#id..........................................................................................................................130
#if expr  #else  #elif  #endif...............................................................................131
#ifdef  #ifndef  #else  #elif  #endif....................................................................132
-ignore_warnings................................................................................................132
#import (options)...............................................................................................133
#include................................................................................................................134
:inline..................................................................................................................135
#int_xxxx............................................................................................................136
Syntax: ..................................................................................................................136
#INT_DEFAULT......................................................................................................140
__line__................................................................................................................140
#list.......................................................................................................................141
#line.....................................................................................................................141
#locate ................................................................. 142
#module ............................................................... 143
#nolist ................................................................. 144
#ocs ...................................................................... 144
#opt ..................................................................... 145
#org ...................................................................... 145
#pin_select ........................................................... 147
__pcd__ ................................................................. 151
#pragma ................................................................. 151
#recursive ............................................................. 153
#reserve ............................................................... 153
#from ................................................................. 154
#rom ................................................................. 155
#separate ............................................................... 155
#serialize ............................................................. 156
#task ................................................................. 158
__time__ ................................................................. 159
#type ................................................................. 160
#undef ................................................................. 162
__unicode ............................................................. 162
#use capture .......................................................... 163
#use delay ............................................................. 165
#use dynamic_memory .................................................. 166
#use fast_io ........................................................... 167
#use fixed_io .......................................................... 167
#use i2c ............................................................... 168
#use profile() .......................................................... 170
#use pwm() ............................................................. 170
#use rs232 .............................................................. 172
#use rtos .............................................................. 178
#use spi ............................................................... 179
#use standard_io ...................................................... 181
#use timer ............................................................ 182
#use touchpad .......................................................... 184
#warning ............................................................... 185
#word ................................................................. 185
#zero_ram .............................................................. 186

Built-in Functions...................................................... 189

BUILT-IN FUNCTIONS ..................................................... 189
abs( ) ................................................................. 189
sin( ) cos( ) tan( ) asin( ) acos( ) atan( ) sinh( ) cosh( ) tanh( ) atan2( ) ........................................ 190
adc_done( ) adc_done2() ................................................ 191
adc_read() ............................................................... 192
adc_status() ............................................................. 193
adc_write() ............................................................. 193
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay_cycles()</td>
<td>227</td>
</tr>
<tr>
<td>delay_ms()</td>
<td>228</td>
</tr>
<tr>
<td>delay_us()</td>
<td>229</td>
</tr>
<tr>
<td>disable_interruptions()</td>
<td>230</td>
</tr>
<tr>
<td>disable_pwm1_interrupt( )</td>
<td>231</td>
</tr>
<tr>
<td>disable_pwm2_interrupt( )</td>
<td>231</td>
</tr>
<tr>
<td>disable_pwm3_interrupt( )</td>
<td>231</td>
</tr>
<tr>
<td>disable_pwm4_interrupt( )</td>
<td>231</td>
</tr>
<tr>
<td>disable_pwm5_interrupt( )</td>
<td>231</td>
</tr>
<tr>
<td>disable_pwm6_interrupt( )</td>
<td>231</td>
</tr>
<tr>
<td>div( )</td>
<td>232</td>
</tr>
<tr>
<td>dma_start()</td>
<td>233</td>
</tr>
<tr>
<td>dma_status()</td>
<td>234</td>
</tr>
<tr>
<td>enable_interruptions( )</td>
<td>234</td>
</tr>
<tr>
<td>erase_program_memory</td>
<td>235</td>
</tr>
<tr>
<td>enable_pwm1_interrupt( )</td>
<td>236</td>
</tr>
<tr>
<td>enable_pwm2_interrupt( )</td>
<td>236</td>
</tr>
<tr>
<td>enable_pwm3_interrupt( )</td>
<td>236</td>
</tr>
<tr>
<td>enable_pwm4_interrupt( )</td>
<td>236</td>
</tr>
<tr>
<td>enable_pwm5_interrupt( )</td>
<td>236</td>
</tr>
<tr>
<td>enable_pwm6_interrupt( )</td>
<td>236</td>
</tr>
<tr>
<td>exp( )</td>
<td>237</td>
</tr>
<tr>
<td>ext_int_edge()</td>
<td>238</td>
</tr>
<tr>
<td>fabs( )</td>
<td>239</td>
</tr>
<tr>
<td>getc()</td>
<td>239</td>
</tr>
<tr>
<td>getch()</td>
<td>239</td>
</tr>
<tr>
<td>getchar()</td>
<td>239</td>
</tr>
<tr>
<td>fgets( )</td>
<td>240</td>
</tr>
<tr>
<td>floor( )</td>
<td>241</td>
</tr>
<tr>
<td>fmod( )</td>
<td>242</td>
</tr>
<tr>
<td>printf( )</td>
<td>242</td>
</tr>
<tr>
<td>fprintf( )</td>
<td>242</td>
</tr>
<tr>
<td>putchar( )</td>
<td>243</td>
</tr>
<tr>
<td>puts( )</td>
<td>244</td>
</tr>
<tr>
<td>puts( )</td>
<td>245</td>
</tr>
<tr>
<td>free( )</td>
<td>246</td>
</tr>
<tr>
<td>frexp( )</td>
<td>246</td>
</tr>
<tr>
<td>scanf( )</td>
<td>247</td>
</tr>
<tr>
<td>get_capture()</td>
<td>250</td>
</tr>
<tr>
<td>get_capture()</td>
<td>251</td>
</tr>
<tr>
<td>get_capture_ccp1()</td>
<td>251</td>
</tr>
<tr>
<td>get_capture_ccp2()</td>
<td>252</td>
</tr>
<tr>
<td>get_capture_ccp3()</td>
<td>252</td>
</tr>
<tr>
<td>get_capture_ccp4()</td>
<td>252</td>
</tr>
<tr>
<td>get_capture_ccp5()</td>
<td>252</td>
</tr>
<tr>
<td>get_capture32_ccp1()</td>
<td>253</td>
</tr>
<tr>
<td>get_capture32_ccp2()</td>
<td>253</td>
</tr>
<tr>
<td>get_capture32_ccp3()</td>
<td>253</td>
</tr>
<tr>
<td>get_capture32_ccp4()</td>
<td>253</td>
</tr>
<tr>
<td>get_capture32_ccp5()</td>
<td>253</td>
</tr>
<tr>
<td>get_capture_event()</td>
<td>254</td>
</tr>
<tr>
<td>get_capture_time()</td>
<td>254</td>
</tr>
<tr>
<td>get_capture32()</td>
<td>255</td>
</tr>
<tr>
<td>get_hspwm_capture()</td>
<td>256</td>
</tr>
<tr>
<td>get_motor_pwm_count()</td>
<td>256</td>
</tr>
<tr>
<td>get_nco_accumulator( )</td>
<td>257</td>
</tr>
<tr>
<td>get_nco_inc_value( )</td>
<td>257</td>
</tr>
<tr>
<td>get_ticks()</td>
<td>258</td>
</tr>
<tr>
<td>get_timerA()</td>
<td>258</td>
</tr>
<tr>
<td>get_timerB()</td>
<td>259</td>
</tr>
<tr>
<td>get_timerx()</td>
<td>259</td>
</tr>
</tbody>
</table>
get_timerxy() ................................................................. 260
get_timer_ccp1() get_timer_ccp2() get_timer_ccp3() get_timer_ccp4()  get_timer_ccp5() ............................................................. 261
get_tris_x() .................................................................... 262
getenv() .......................................................................... 263
goto_address() .................................................................. 268
high_speed_adc_done() ....................................................... 268
i2c_init() ........................................................................... 269
i2c_isr_state() .................................................................... 270
i2c_poll() ............................................................................. 271
i2c_read() .......................................................................... 271
i2c_slaveaddr() .................................................................. 272
i2c_speed() ........................................................................... 273
i2c_start() ........................................................................... 274
i2c_stop() ............................................................................ 275
i2c_write() ........................................................................... 275
input() ................................................................................ 276
input_change_x() ................................................................ 277
input_state() .................................................................... 278
input_x() ............................................................................. 279
interrupt_active() .............................................................. 279
isalnum(char) isalpha(char) iscntrl(x) isdigit(char) isgraph(x) islower(char) isspace(char) isupper(char) isxdigit(char) isprint(x) ispunct(x) 280
isamong() ........................................................................... 281
itoa() ............................................................................... 282
kbhit() ............................................................................... 283
label_address() ................................................................. 284
labs() ................................................................................ 284
ldexp() ............................................................................... 285
log() ................................................................................ 286
log10() .............................................................................. 286
longjmp() ............................................................................ 287
make8() ............................................................................. 288
make16() .......................................................................... 288
make32() ........................................................................... 289
malloc() ............................................................................ 290
memcpy() memmove() ........................................................... 290
memset() ............................................................................ 291
modf() ............................................................................... 292
_mul() ............................................................................... 292
nargs() .............................................................................. 293
offsetof() offsetofbit() ....................................................... 294
output_x() .......................................................................... 295
output_bit() ..................................................................... 296
rcv_buffer_bytes() .................................................. 334
rcv_buffer_full() .................................................. 335
read_adc()  read_adc2() ........................................... 335
read_configuration_memory() ..................................... 337
read_eeprom() ...................................................... 337
read_extended_ram() .............................................. 338
read_program_memory() ........................................... 339
read_high_speed_adc() ............................................ 339
read_rom_memory() ................................................ 341
read_sd_adc() ...................................................... 342
realloc() ............................................................ 343
release_io() ......................................................... 344
reset_cpu() .......................................................... 344
restart_cause() ...................................................... 345
restart_wdt() ........................................................ 346
rotate_left() ........................................................ 347
rotate_right() ....................................................... 347
rtc_alarm_read() .................................................... 348
rtc_alarm_write() ................................................... 349
rtc_read() ............................................................ 349
rtc_write() .......................................................... 350
rtosAwait() .......................................................... 351
rtos_disable() ....................................................... 351
rtos_enable() ......................................................... 352
rtos_msg_poll() ...................................................... 352
rtos_msg_read() ...................................................... 353
rtos_msg_send() ...................................................... 353
rtos_overrun() ...................................................... 354
rtos_run() ............................................................ 355
rtos_signal() ........................................................ 355
rtos_stats() ......................................................... 356
rtos_terminate() ..................................................... 357
rtos_wait() ........................................................... 357
rtos_yield() .......................................................... 358
set_adc_channel()  set_adc_channel2() ........................... 359
set_adc_trigger() .................................................... 360
set_analog_pins() .................................................... 360
scanf() ............................................................... 361
set_ccp1_compare_time()  set_ccp2_compare_time() .............. 361
set_ccp3_compare_time()  set_ccp4_compare_time() .............. 361
set_ccp5_compare_time() ........................................... 365
set_cog_blanking() .................................................. 366
set_cog_dead_band() ................................................ 367
set_cog_phase() ..................................................... 368
set_compare_time() ................................................ 368
Table of Contents

set_compare_time( ) .................................................................369
set_dedicated_adc_channel( ) ..................................................370
set_hspwm_override( ) ..........................................................371
set_hspwm_phase( ) ..............................................................372
set_input_level_x() .............................................................372
set_motor_pwm_duty( ) ..........................................................373
set_motor_pwm_event( ) .........................................................374
set_motor_unit( ) .................................................................375
set_nco_inc_value( ) .............................................................375
set_pullup() ........................................................................377
set_rtc ................................................................................378
set_set_pwm1_duty( ) set_pwm2_duty( ) set_pwm3_duty() ........379
set_pwm1_duty( ) set_pwm4_duty( ) set_pwm5_duty() ............379
set_pwm1_offset() set_pwm2_offset() set_pwm3_offset() ......380
set_pwm4_offset() set_pwm5_offset() set_pwm6_offset() ......380
set_pwm1_period() set_pwm2_period() set_pwm3_period() ....381
set_pwm4_period() set_pwm5_period() set_pwm6_period() ....381
set_pwm1_phase() set_pwm2_phase() set_pwm3_phase() ......382
set_pwm4_phase() set_pwm5_phase() set_pwm6_phase() ......382
set_open_drain_x() ...............................................................383
set_rtc set_timer0() set_timer1() set_timer2() set_timer3() ....383
set_timer4() set_timer5() ........................................................384
set_ticks() ........................................................................385
setup_sd_adc_calibration() ....................................................385
set_sd_adc_channel() ............................................................386
set_timerA() ........................................................................387
set_timerB() ........................................................................387
set_timerx() ........................................................................388
set_timerxy() ......................................................................388
set_rtc set_timer0() set_timer1() set_timer2() set_timer3() ....388
set_timer4() set_timer5() ........................................................389
set_timer_ccp1() set_timer_ccp2() set_timer_ccp3() ..........389
set_timer_ccp4() set_timer_ccp5() ..........................................390
set_timer_period_ccp1() set_timer_period_ccp2() ............390
set_timer_period_ccp3() set_timer_period_ccp4() ..........391
set_timer_period_ccp5() ........................................................391
set_tris_x() ........................................................................392
set_uart_speed() .................................................................393
setmp() ..............................................................................394
setup_adc(mode) setup_adc2(mode) ........................................395
setup_adc_ports() setup_adc_ports2() ..................................396
setup_adc_reference() ..........................................................397
setup_at() ...........................................................................398
setup_capture() .................................................................399
setup_ccp1()  setup_ccp2()  setup_ccp3()  setup_ccp4()
setup_ccp5()  setup_ccp6() .......................................................... 399
setup_clc1()  setup_clc2()  setup_clc3()  setup_clc4() .................... 402
setup_comparator() ................................................................. 403
setup_compare() .................................................................. 404
setup_crc(mode) .................................................................. 404
setup_cog() ......................................................................... 405
setup_crc() ......................................................................... 406
setup_cwg() ........................................................................ 407
setup_dac() ........................................................................ 408
setup_dci() .......................................................................... 409
setup_dedicated_adc() ................................................................ 410
setup_dma() ........................................................................... 411
setup_high_speed_adc() ................................................................ 412
setup_high_speed_adc_pair() .......................................................... 412
setup_hspwm_blanking() ................................................................ 413
setup_hspwm_chop_clock() .............................................................. 415
setup_hspwm_trigger() ................................................................. 415
setup_hspwm_unit() .................................................................. 416
setup_hspwm()  setup_hspwm_secondary() .................................................. 418
setup_hspwm_unit_chop_clock() ......................................................... 419
setup_low_volt_detect() ................................................................. 420
setup_motor_pwm() ................................................................ 420
setup_oscillator() ...................................................................... 421
setup_pga() .............................................................................. 422
setup_pid() ............................................................................. 423
setup_pmp(option,address_mask) ......................................................... 424
setup_psmc() ........................................................................ 425
setup_power_pwm_pins() ................................................................ 427
setup_psp(option,address_mask) ......................................................... 428
setup_pwm1()  setup_pwm2()  setup_pwm3()  setup_pwm4() .................. 429
setup_qei() ............................................................................ 430
setup_rtc() ............................................................................ 431
setup rtc_alarm() ..................................................................... 431
setup_sd_adc() ......................................................................... 432
setup_smtx() ........................................................................... 433
setup_spi()  setup_spi2() ................................................................. 434
setup_timerx() ........................................................................ 435
setup_timer_A() ........................................................................ 436
setup_timer_B() ........................................................................ 437
setup_timer_0() ........................................................................ 438
setup_timer_1() ........................................................................ 438
setup_timer_2() ........................................................................ 439
setup_timer_3() ........................................................................ 440
setup_timer_4() ........................................................................ 441
PCD

PCD is a C Compiler for Microchip's 24bit opcode family of microcontrollers, which include the dsPIC30, dsPIC33 and PIC24 families. The compiler is specifically designed to meet the unique needs of the dsPIC® microcontroller. This allows developers to quickly design applications software in a more readable, high-level language.

The compiler can efficiently implement normal C constructs, input/output operations, and bit twiddling operations. All normal C data types are supported along with special built-in functions to perform common functions in the MPU with ease.

Extended constructs like bit arrays, multiple address space handling and effective implementation of constant data in Rom make code generation very effective.
Technical Support

Compiler, software, and driver updates are available to download at: http://www.ccsinfo.com/download

Compilers come with 30 or 60 days of download rights with the initial purchase. One year maintenance plans may be purchased for access to updates as released.

The intent of new releases is to provide up-to-date support with greater ease of use and minimal, if any, transition difficulty.

To ensure any problem that may occur is corrected quickly and diligently, it is recommended to send an email to: support@ccsinfo.com or use the Technical Support Wizard in PCW. Include the version of the compiler, an outline of the problem and attach any files with the email request. CCS strives to answer technical support timely and thoroughly.

Technical Support is available by phone during business hours for urgent needs or if email responses are not adequate. Please call 262-522-6500 x32.

Directories

The compiler will search the following directories for Include files.

- Directories listed on the command line
- Directories specified in the .CCSPJT file
- The same directory as the source.directories in the ccsc.ini file

By default, the compiler files are put in C:\Program Files\PICC and the example programs are in \PICC\EXAMPLES. The include files are in PICC\drivers. The device header files are in PICC\devices.

The compiler itself is a DLL file. The DLL files are in a DLL directory by default in \PICC\DLL.

It is sometimes helpful to maintain multiple compiler versions. For example, a project was tested with a specific version, but newer projects use a newer version. When installing the
compiler you are prompted for what version to keep on the PC. IDE users can change versions using Help>about and clicking "other versions." Command Line users use start>all programs>PIC-C>compiler version.

Two directories are used outside the PICC tree. Both can be reached with start>all programs>PIC-C.

1.) A project directory as a default location for your projects. By default put in "My Documents." This is a good place for VISTA and up.

2.) User configuration settings and PCWH loaded files are kept in %APPDATA%\PICC

**File Formats**

<table>
<thead>
<tr>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.c</td>
<td>This is the source file containing user C source code.</td>
</tr>
<tr>
<td>.h</td>
<td>These are standard or custom header files used to define pins, register, register bits, functions and preprocessor directives.</td>
</tr>
<tr>
<td>.pjt</td>
<td>This is the older pre-Version 5 project file which contains information related to the project.</td>
</tr>
<tr>
<td>.ccspjt</td>
<td>This is the project file which contains information related to the project.</td>
</tr>
<tr>
<td>.lst</td>
<td>This is the listing file which shows each C source line and the associated assembly code generated for that line. The elements in the .LST file may be selected in PCW under Options&gt;Project&gt;Output Files</td>
</tr>
<tr>
<td>.sym</td>
<td>This is the symbol map which shows each register location and what program variables are stored in each location.</td>
</tr>
</tbody>
</table>

The elements in the .LST file may be selected in PCW under Options>Project>Output Files

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCS Basic with Opcodes</td>
<td>Includes the HEX opcode for each instruction</td>
</tr>
<tr>
<td>Old Standard Symbolic</td>
<td>Shows variable names instead of addresses</td>
</tr>
<tr>
<td>File Extension</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>.sta</td>
<td>The statistics file shows the RAM, ROM, and STACK usage. It provides information on the source codes structural and textual complexities using Halstead and McCabe metrics.</td>
</tr>
<tr>
<td>.tre</td>
<td>The tree file shows the call tree. It details each function and what functions it calls along with the ROM and RAM usage for each function.</td>
</tr>
<tr>
<td>.hex</td>
<td>The compiler generates standard HEX files that are compatible with all programmers. The compiler can output 8-bit hex, 16-bit hex, and binary files.</td>
</tr>
<tr>
<td>.cof</td>
<td>This is a binary containing machine code and debugging information. The debug files may be output as Microchip .COD file for MPLAB 1-5, Advanced Transdata .MAP file, expanded .COD file for CCS debugging or MPLAB 6 and up .xx .COF file. All file formats and extensions may be selected via Options File Associations option in Windows IDE.</td>
</tr>
<tr>
<td>.cod</td>
<td>This is a binary file containing debug information.</td>
</tr>
<tr>
<td>.rtf</td>
<td>The output of the Documentation Generator is exported in a Rich Text File format which can be viewed using the RTF editor or Wordpad.</td>
</tr>
<tr>
<td>.rvf</td>
<td>The Rich View Format is used by the RTF Editor within the IDE to view the Rich Text File.</td>
</tr>
<tr>
<td>.dgr</td>
<td>The .DGR file is the output of the flowchart maker.</td>
</tr>
<tr>
<td>.esym</td>
<td>These files are generated for the IDE users. The file contains Identifiers and Comment information. This data can be used for automatic documentation generation and for the IDE helpers.</td>
</tr>
<tr>
<td>.xsym</td>
<td></td>
</tr>
<tr>
<td>.o</td>
<td>Relocatable object file</td>
</tr>
<tr>
<td>.osym</td>
<td>This file is generated when the compiler is set to export a relocatable object file. This file is a .sym file for just the one unit.</td>
</tr>
<tr>
<td>.err</td>
<td>Compiler error file</td>
</tr>
<tr>
<td>.ccsload</td>
<td>used to link Windows 8 apps to CCSLoad</td>
</tr>
<tr>
<td>.ccssio</td>
<td>used to link Windows 8 apps to Serial Port Monitor</td>
</tr>
</tbody>
</table>
Program Syntax

Invoking the Command Line Compiler
The command line compiler is invoked with the following command:

CCSC

[options]

[cfilename]

Valid options:
+FB
Select PCB (12 bit)
+FM
Select PCM (14 bit)
+FH
Select PCH (PIC18XXX)
+Yx
Optimization level x (0-9)

-D
+DS
+DM
+DC
+DF

+FS
+ES
+T
+A
+EW

Select SXC (SX)
Standard error file
Create call tree (.TRE)
Create stats file (.STA)
Show warning messages

+EO
-T
-A
-EW
-E

+EA

Show all error messages
and all warnings

+EX

Do not create debug file
Standard .COD format debug file
.MAP format debug file
Expanded .COD format debug file
Enables the output of an COFF debug
file.
Old error file format
Do not generate a tree file
Do not create stats file (.STA)
Suppress warnings (use with +EA)
Only show first error
Error/warning message format uses
GCC's "brief format" (compatible with
GCC editor environments)

The xxx in the following are optional. If included it sets the file extension:
+LNxxx
Normal list file
+O8xxx 8-bit Intel HEX output file
+LSxxx
MPASM format list
+OWxxx 16-bit Intel HEX output file
file
+LOxxx
Old MPASM list file
+OBxxx
Binary output file
+LYxxx
Symbolic list file
-O
Do not create object file
-L
Do not create list file
+P
+Pxx
+PN
+PE

Keep compile status window up after compile
Keep status window up for xx seconds after compile
Keep status window up only if there are no errors
Keep status window up only if there are errors

+Z
+DF
I+="..."

Keep scratch files on disk after compile
COFF Debug file
Same as I="..." Except the path list is appended to the current list

I="..."

Set include directory search path, for example:
I="c:\picc\examples;c:\picc\myincludes"
If no I= appears on the command line the .PJT file will be used to supply
the include file paths.

19


<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-P</td>
<td>Close compile window after compile is complete</td>
</tr>
<tr>
<td>+M</td>
<td>Generate a symbol file (.SYM)</td>
</tr>
<tr>
<td>-M</td>
<td>Do not create symbol file</td>
</tr>
<tr>
<td>+J</td>
<td>Create a project file (.PJT)</td>
</tr>
<tr>
<td>-J</td>
<td>Do not create PJT file</td>
</tr>
<tr>
<td>+ICD</td>
<td>Compile for use with an ICD</td>
</tr>
<tr>
<td>#xxx=&quot;yyy&quot;</td>
<td>Set a global #define for id xxx with a value of yyy, example: #debug=&quot;true&quot;</td>
</tr>
<tr>
<td>+Gxxx=&quot;yyy&quot;</td>
<td>Same as #xxx=&quot;yyy&quot;</td>
</tr>
<tr>
<td>+?</td>
<td>Brings up a help file</td>
</tr>
<tr>
<td>-?</td>
<td>Same as +?</td>
</tr>
<tr>
<td>+STDOUT</td>
<td>Outputs errors to STDOUT (for use with third party editors)</td>
</tr>
<tr>
<td>+SETUP</td>
<td>Install CCSC into MPLAB (no compile is done)</td>
</tr>
</tbody>
</table>
| sourceline= | Allows a source line to be injected at the start of the source file. Example: CCSC +FM myfile.c sourceline="#include <16F887.h>"
| +V     | Show compiler version (no compile is done) |
| +Q     | Show all valid devices in database (no compile is done) |

A / character may be used in place of a + character. The default options are as follows:

- FM +ES +J +DC +Y9 -T -A +M +L +St +O8hex -P -Z

If @filename appears on the CCSC command line, command line options will be read from the specified file. Parameters may appear on multiple lines in the file.

If the file CCSC.INI exists in the same directory as CCSC.EXE, then command line parameters are read from that file before they are processed on the command line.

Examples:

```plaintext
CCSC +FM C:\PICSTUFF\TEST.C
CCSC +FM +P +T TEST.C
```
PCW Overview

The PCW IDE provides the user an easy to use editor and environment for developing microcontroller applications. The IDE comprises of many components, which are summarized below. For more information and details, use the Help>PCW in the compiler.

Many of these windows can be re-arranged and docked into different positions.
Menu

All of the IDE's functions are on the main menu. The main menu is divided into separate sections, click on a section title ('Edit', 'Search', etc) to change the section. Double clicking on the section, or clicking on the chevron on the right, will cause the menu to minimize and take less space.

Editor Tabs

All of the open files are listed here. The active file, which is the file currently being edited, is given a different highlight than the other files. Clicking on the X on the right closes the active file. Right clicking on a tab gives a menu of useful actions for that file.

Slide Out Windows

'Files' shows all the active files in the current project. 'Projects' shows all the recent projects worked on. 'Identifiers' shows all the variables, definitions, prototypes and identifiers in your current project.
Editor

The editor is the main work area of the IDE and the place where the user enters and edits source code. Right clicking in this area gives a menu of useful actions for the code being edited.

```
100 #if defined(USB_HW_CCSTMPIC18F4550)
101 #include <18F4550.h>
102 #fuses HSPLL, NOWDT, NOPROTECT, NONVP, NODEBUG, USBDIV, PLL5, CFUDIV1, VREGEN
103
104 #use delay(clock=48000000)
105
106 //leds ordered from bottom to top
107 #DEFINE LED1 PIN_A5 //green
```

Debugging Windows

Debugger control is done in the debugging windows. These windows allow you set breakpoints, single step, watch variables and more.
Status Bar

The status bar gives the user helpful information like the cursor position, project open and file being edited.

Output Messages

Output messages are displayed here. This includes messages from the compiler during a build, messages from the programmer tool during programming or the results from find and searching.
Overall Structure

A program is made up of the following four elements in a file:

Comment
Pre-Processor Directive
Data Definition
Function Definition
  Statements
  Expressions

Every C program must contain a main function which is the starting point of the program execution. The program can be split into multiple functions according to their purpose and the functions could be called from main or the sub-functions. In a large project functions can also be placed in different C files or header files that can be included in the main C file to group the related functions by their category. CCS C also requires to include the appropriate device file using #include directive to include the device specific functionality. There are also some preprocessor directives like #fuses to specify the fuses for the chip and #use delay to specify the clock speed. The functions contain the data declarations, definitions, statements and expressions. The compiler also provides a large number of standard C libraries as well as other device drivers that can be included and used in the programs. CCS also provides a large number of built-in functions to access the various peripherals included in the PIC microcontroller.

Comment

Comments – Standard Comments
A comment may appear anywhere within a file except within a quoted string. Characters between /* and */ are ignored. Characters after a // up to the end of the line are ignored.

Comments for Documentation Generator
The compiler recognizes comments in the source code based on certain markups. The compiler recognizes these special types of comments that can be later exported for use in
the documentation generator. The documentation generator utility uses a user selectable template to export these comments and create a formatted output document in Rich Text File Format. This utility is only available in the IDE version of the compiler. The source code markups are as follows.

**Global Comments**
These are named comments that appear at the top of your source code. The comment names are case sensitive and they must match the case used in the documentation template.
For example:
//*/PURPOSE This program implements a Bootloader.
//*/AUTHOR John Doe

A '/*' followed by an * will tell the compiler that the keyword which follows it will be the named comment. The actual comment that follows it will be exported as a paragraph to the documentation generator.
Multiple line comments can be specified by adding a : after the *, so the compiler will not concatenate the comments that follow. For example:
/**:CHANGES
   05/16/06 Added PWM loop
   05/27.06 Fixed Flashing problem
*/

**Variable Comments**
A variable comment is a comment that appears immediately after a variable declaration. For example:
int seconds; // Number of seconds since last entry
long day, // Current day of the month, /* Current Month */
long year; // Year

**Function Comments**
A function comment is a comment that appears just before a function declaration. For example:
// The following function initializes outputs
void function_foo()
{
    init_outputs();
}
Function Named Comments
The named comments can be used for functions in a similar manner to the Global Comments. These comments appear before the function, and the names are exported as-is to the documentation generator.
For example:
```c
//*PURPOSE This function displays data in BCD format
void display_BCD( byte n)
{
    display_routine();
}
```

Trigraph Sequences

The compiler accepts three character sequences instead of some special characters not available on all keyboards as follows:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>??=</td>
<td>#</td>
</tr>
<tr>
<td>??(</td>
<td>[</td>
</tr>
<tr>
<td>??/</td>
<td>\</td>
</tr>
<tr>
<td>??)</td>
<td>]</td>
</tr>
<tr>
<td>??'</td>
<td>^</td>
</tr>
<tr>
<td>??&lt;</td>
<td>{</td>
</tr>
<tr>
<td>??!</td>
<td>}</td>
</tr>
<tr>
<td>??&gt;</td>
<td>}</td>
</tr>
<tr>
<td>??-</td>
<td>~</td>
</tr>
</tbody>
</table>

Multiple Project Files

When there are multiple files in a project they can all be included using the #include in the main file or the sub-files to use the automatic linker included in the compiler. All the header files, standard libraries and driver files can be included using this method to automatically link them.

For example: if you have main.c, x.c, x.h, y.c,y.h and z.c and z.h files in your project, you can say in:
# Multiple Compilation Units

Multiple Compilation Units are only supported in the IDE compilers, PCW, PCWH, PCHWD and PCDIDE. When using multiple compilation units, care must be given that pre-processor commands that control the compilation are compatible across all units. It is recommended that directives such as #FUSES, #USE and the device header file all put in an include file included by all units. When a unit is compiled it will output a relocatable object file (*.o) and symbol file (*.osym).

There are several ways to accomplish this with the CCS C Compiler. All of these methods and example projects are included in the MCU.zip in the examples directory of the compiler.

## Full Example Program

Here is a sample program with explanation using CCS C to read adc samples over rs232:
This program displays the min and max of 30 A/D samples over the RS-232 interface. The process is repeated forever.

If required configure the CCS prototype card as follows:

- Insert jumper from output of POT to pin A5
- Use a 10K POT to vary the voltage.

Jumpers:

- PCM,PCH: pin C7 to RS232 RX, pin C6 to RS232 TX
- PCD: none

This example will work with the PCM, PCH, and PCD compilers.

The following conditional compilation lines are used to include a valid device for each compiler. Change the device, clock and RS232 pins for your hardware if needed.

(C) Copyright 1996,2007 Custom Computer Services

This source code may only be used by licensed users of the CCS C compiler. It may only be distributed to other licensed users of the CCS C compiler. No other use, reproduction or distribution is permitted without written permission. Derivative programs created using this software in object code form are not restricted in any way.

#define __PCM__
// Preprocessor directive that chooses
#include <16F877.h>
// Preprocessor directive that selects
#define HS,NOWDT,NOPROTECT,NOLVP
// Preprocessor directive that defines
#define delay(clock=20000000)
// Preprocessor directive that specifies clock speed
#define rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7) // Preprocessor directive that includes

#define __PCH__
#include <18F452.h>
define HS,NOWDT,NOPROTECT,NOLVP
define delay(clock=20000000)
define rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7)
#device ADC=8
#use delay(clock=20000000)
#use rs232(baud=9600, UART1A)
#endif

void main() {
    unsigned int8 i, value, min, max;
    printf("Sampling:"); // Printf function included in RS232
    setup_adc_ports(AN0);
    // library
    setup_adc(ADC_CLOCK_INTERNAL); // Built-in A/D setup function
    set_adc_channel(0); // Built-in A/D setup function
    do {
        min=255;
        max=0;
        for (i=0; i<=30; ++i) {
            delay_ms(100); // Built-in delay function
            value = read_adc(); // Built-in A/D read function
            if (value<min)
                min=value;
            if (value>max)
                max=value;
        }
        printf("\r\nMin: %2X Max: %2X\n",min,max);
    } while (TRUE);
}
## STATEMENTS

### Statements

<table>
<thead>
<tr>
<th>STATEMENT</th>
<th>Example</th>
</tr>
</thead>
</table>
| **if** (expr) stmt; [else stmt:] | if (x==25)  
    x=0;  
    else  
    x=x+1; |
| **while** (expr) stmt;         | while (get_rtcc() !=0)  
    putc('n'); |
| **do** stmt **while** (expr);  | do {  
    putc(c=getc());  
} while (c!=0); |
| **for** (expr1;expr2;expr3) stmt; | for (i=1; i<=10; ++i)  
    printf("%u\n",i); |
| **switch** (expr) {  
    case expr: stmt; //one or more case  
    [default:stmt]  
    ... }            | switch (cmd) {  
    case 0:  printf("cmd 0");break;  
    case 1:  printf("cmd 1");break;  
    default:  printf("bad cmd");break; |
| **return** [expr];            | return (5); |
| **goto** label;               | goto loop; |
| label: stmt;                  | loop: i++; |
| **break**;                    | break; |
| **continue**;                 | continue; |
| **expr**;                     | i=1;  
| ;                              | |
| {[stmt]}                      | {a=1;  
    b=1;} |
| Zero or more declaration;     | int i; |

Note: Items in [ ] are optional
if

if-else
The if-else statement is used to make decisions.
The syntax is:

    if (expr)
        stmt-1;
    [else
        stmt-2;]

The expression is evaluated; if it is true stmt-1 is done. If it is false then stmt-2 is done.

else-if
This is used to make multi-way decisions.
The syntax is:

    if (expr)
        stmt;
    [else if (expr)
        stmt;]
    ...
    [else
        stmt;]

The expressions are evaluated in order; if any expression is true, the statement associated with it is executed and it terminates the chain. If none of the conditions are satisfied the last else part is executed.

Example:
    if (x==25)
        x=1;
    else
        x=x+1;

Also See: Statements
while

While is used as a loop/iteration statement. The syntax is:

```
while (expr)
  statement
```

The expression is evaluated and the statement is executed until it becomes false in which case the execution continues after the statement.

Example:
```
while (get_rtcc() != 0)
  putc('n');
```

Also See: Statements

do-while

do-while: Differs from while and for loop in that the termination condition is checked at the bottom of the loop rather than at the top and so the body of the loop is always executed at least once. The syntax is:

```
do
  statement
while (expr);
```

The statement is executed; the expr is evaluated. If true, the same is repeated and when it becomes false the loop terminates.

Also See: Statements, While

for

For is also used as a loop/iteration statement. The syntax is:

```
for (expr1; expr2; expr3)
  statement
```
The expressions are loop control statements. expr1 is the initialization, expr2 is the termination check and expr3 is re-initialization. Any of them can be omitted.

Example:
```
for (i=1;i<=10;++i)
   printf("%u\r\n",i);
```

Also See: Statements

**switch**

**Switch** is also a special multi-way decision maker.
The syntax is
```
switch (expr) {
   case const1: stmt sequence;
       break;
   ...
   [default:stmt]
}
```

This tests whether the expression matches one of the constant values and branches accordingly.
If none of the cases are satisfied the default case is executed. The break causes an immediate exit, otherwise control falls through to the next case.

Example:
```
switch (cmd) {
   case 0:printf("cmd 0");
       break;
   case 1:printf("cmd 1");
       break;
   default:printf("bad cmd");
       break; }
```

Also See: Statements
**return**

A `return` statement allows an immediate exit from a switch or a loop or function and also returns a value.

The syntax is:

```
return(expr);
```

Example:
```
return (5);
```

Also See: [Statements](#)

**goto**

The `goto` statement cause an unconditional branch to the label.

The syntax is:
```
goto label;
```

A label has the same form as a variable name, and is followed by a colon. The goto's are used sparingly, if at all.

Example:
```
goto loop;
```

Also See: [Statements](#)

**label**

The label a goto jumps to.

The syntax is:

```
label: stmt;
```
Example:
loop: i++;

Also See: Statements

**break**

**break.**
The break statement is used to exit out of a control loop. It provides an early exit from while, for, do and switch.
The syntax is

```
break;
```

It causes the innermost enclosing loop (or switch) to be exited immediately.

Example:
break;

Also See: Statements

**continue**

The **continue** statement causes the next iteration of the enclosing loop(While, For, Do) to begin.
The syntax is:

```
continue;
```

It causes the test part to be executed immediately in case of do and while and the control passes the re-initialization step in case of for.

Example:
continue;

Also See: Statements
**expr**

The syntax is:
expr;

Example:
i=1;

Also See: Statements

;

Statement: ;

Example:
;

Also See: Statements

**stmt**

Zero or more semi-colon separated.
The syntax is:

{{stmt}}

Example:
{a=1;
 b=1; }

Also See: Statements
## Expressions

### Constants

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Decimal</td>
</tr>
<tr>
<td>123L</td>
<td>Forces type to &amp; long (UL also allowed)</td>
</tr>
<tr>
<td>123LL</td>
<td>Forces type to &amp;; 64 for PCD</td>
</tr>
<tr>
<td>0123</td>
<td>Octal</td>
</tr>
<tr>
<td>0x123</td>
<td>Hex</td>
</tr>
<tr>
<td>0b010010</td>
<td>Binary</td>
</tr>
<tr>
<td>123.456</td>
<td>Floating Point</td>
</tr>
<tr>
<td>123F</td>
<td>Floating Point (FL also allowed)</td>
</tr>
<tr>
<td>123.4E-5</td>
<td>Floating Point in scientific notation</td>
</tr>
<tr>
<td>'x'</td>
<td>Character</td>
</tr>
<tr>
<td>'\010'</td>
<td>Octal Character</td>
</tr>
<tr>
<td>'\xA5'</td>
<td>Hex Character</td>
</tr>
</tbody>
</table>
| '\c'       | Special Character. Where c is one of: 
  \n  Line Feed - Same as \x0a 
  \r  Return Feed - Same as \x0d 
  \t  TAB - Same as \x09 
  \b  Backspace - Same as \x08 
  \f  Form Feed - Same as \x0c 
  \a  Bell - Same as \x07 
  \v  Vertical Space - Same as \x0b 
  \?  Question Mark - Same as \x3f 
  \'  Single Quote - Same as \x22 |
Double Quote - Same as `\x22`
A Single Backslash - Same as `\x5c`
"abcdef" String (null is added to the end)

## Identifiers

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCDE</td>
<td>Up to 32 characters beginning with a non-numeric. Valid characters are A-Z, 0-9 and _ (underscore). By default not case sensitive. Use #CASE to turn on.</td>
</tr>
<tr>
<td>ID[X]</td>
<td>Single Subscript</td>
</tr>
<tr>
<td>ID[X][X]</td>
<td>Multiple Subscripts</td>
</tr>
<tr>
<td>ID.ID</td>
<td>Structure or union reference</td>
</tr>
<tr>
<td>ID-&gt;ID</td>
<td>Structure or union reference</td>
</tr>
</tbody>
</table>

## Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition Operator</td>
</tr>
<tr>
<td>+=</td>
<td>Addition assignment operator, x+=y, is the same as x=x+y</td>
</tr>
<tr>
<td>[ ]</td>
<td>Array subcript operator</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Bitwise and assignment operator, x&amp;=y, is the same as x=x&amp;y</td>
</tr>
<tr>
<td>&amp;</td>
<td>Address operator</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and operator</td>
</tr>
<tr>
<td>^=</td>
<td>Bitwise exclusive or assignment operator, x^=y, is the same as x=x^y</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive or operator</td>
</tr>
<tr>
<td>l=</td>
<td>Bitwise inclusive or assignment operator, xl=y, is the same as x=xly</td>
</tr>
<tr>
<td>l</td>
<td>Bitwise inclusive or operator</td>
</tr>
<tr>
<td>?:</td>
<td>Conditional Expression operator</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>-</td>
<td>Decrement</td>
</tr>
<tr>
<td>/=</td>
<td>Division assignment operator, ( x/=}y ), is the same as ( x=x/y )</td>
</tr>
<tr>
<td>/</td>
<td>Division operator</td>
</tr>
<tr>
<td>==</td>
<td>Equality</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than operator</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to operator</td>
</tr>
<tr>
<td>++</td>
<td>Increment</td>
</tr>
<tr>
<td>*</td>
<td>Indirection operator</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>Left shift assignment operator, ( x&lt;&lt;=}y ), is the same as ( x=x&lt;&lt;y )</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than operator</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Left Shift operator</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to operator</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND operator</td>
</tr>
<tr>
<td>!</td>
<td>Logical negation operator</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>Member operator for structures and unions</td>
</tr>
<tr>
<td>%=</td>
<td>Modules assignment operator ( x%=y ), is the same as ( x=x%y )</td>
</tr>
<tr>
<td>%</td>
<td>Modules operator</td>
</tr>
<tr>
<td>*=</td>
<td>Multiplication assignment operator, ( x*=y ), is the same as ( x=x*y )</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication operator</td>
</tr>
<tr>
<td>~</td>
<td>One's complement operator</td>
</tr>
<tr>
<td>&gt;&gt;&gt;=</td>
<td>Right shift assignment, ( x&gt;&gt;=}y ), is the same as ( x=x&gt;&gt;y )</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift operator</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Structure Pointer operation</td>
</tr>
<tr>
<td>-=</td>
<td>Subtraction assignment operator, ( x-=y ), is the same as ( x=x- y )</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction operator</td>
</tr>
</tbody>
</table>
### sizeof

Determines size in bytes of operand

See also: [Operator Precedence](#)

---

## Operator Precedence

<table>
<thead>
<tr>
<th>PIN DESCENDING PRECEDENCE</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(expr)</code></td>
<td><code>exor++</code></td>
</tr>
<tr>
<td><code>++expr</code></td>
<td><code>expr++</code></td>
</tr>
<tr>
<td><code>!expr</code></td>
<td><code>~expr</code></td>
</tr>
<tr>
<td><code>(type)expr</code></td>
<td><code>*expr</code></td>
</tr>
<tr>
<td><code>expr*expr</code></td>
<td><code>expr/expr</code></td>
</tr>
<tr>
<td><code>expr+expr</code></td>
<td><code>expr-expr</code></td>
</tr>
<tr>
<td><code>expr&lt;&lt;expr</code></td>
<td><code>expr&gt;&gt;expr</code></td>
</tr>
<tr>
<td><code>expr&lt;expr</code></td>
<td><code>expr&lt;=expr</code></td>
</tr>
<tr>
<td><code>expr==expr</code></td>
<td><code>expr!=expr</code></td>
</tr>
<tr>
<td><code>expr&amp;expr</code></td>
<td><code>expr^expr</code></td>
</tr>
<tr>
<td>`expr</td>
<td>expr`</td>
</tr>
<tr>
<td><code>expr &amp;&amp; expr</code></td>
<td><code>expr &amp;&amp; expr</code></td>
</tr>
<tr>
<td><code>expr ? expr : expr</code></td>
<td><code>lvalue = expr</code></td>
</tr>
</tbody>
</table>

(Operators on the same line are equal in precedence)
DATA DEFINITIONS

Data Definitions

This section describes what the basic data types and specifiers are and how variables can be declared using those types. In C all the variables should be declared before they are used. They can be defined inside a function (local) or outside all functions (global). This will affect the visibility and life of the variables.

A declaration consists of a type qualifier and a type specifier, and is followed by a list of one or more variables of that type.

For example:

```c
int a, b, c, d;
mybit e, f;
mybyte g[3][2];
char *h;
colors j;
struct data_record data[10];
static int i;
extern long j;
```

Variables can also be declared along with the definitions of the special types.

For example:

```c
enum colors {red, green=2, blue} i, j, k;  // colors is the
enum type and i, j, k
  // are variables
  // of that type
```

Type Specifiers

Basic Types
### Type Specifiers

<table>
<thead>
<tr>
<th>Type Specifier</th>
<th>Size</th>
<th>Unsigned</th>
<th>Signed</th>
<th>Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>int1</td>
<td>1 bit number</td>
<td>0 to 1</td>
<td>N/A</td>
<td>1/2</td>
</tr>
<tr>
<td>int8</td>
<td>8 bit number</td>
<td>0 to 255</td>
<td>-128 to 127</td>
<td>2-3</td>
</tr>
<tr>
<td>int16</td>
<td>16 bit number</td>
<td>0 to 65535</td>
<td>-32768 to 32767</td>
<td>4-5</td>
</tr>
<tr>
<td>int32</td>
<td>32 bit number</td>
<td>0 to 4294967295</td>
<td>-2147483648 to 2147483647</td>
<td>9-10</td>
</tr>
<tr>
<td>float32</td>
<td>32 bit float</td>
<td>-1.5 x 10^{45} to 3.4 x 10^{38}</td>
<td>7-8</td>
<td></td>
</tr>
</tbody>
</table>

### C Standard Type

<table>
<thead>
<tr>
<th>C Standard Type</th>
<th>Default Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>int1</td>
</tr>
<tr>
<td>char</td>
<td>unsigned int8</td>
</tr>
<tr>
<td>int</td>
<td>int8</td>
</tr>
<tr>
<td>long</td>
<td>int16</td>
</tr>
<tr>
<td>long long</td>
<td>int32</td>
</tr>
<tr>
<td>float</td>
<td>float32</td>
</tr>
<tr>
<td>double</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Note: All types, except float char, by default are un-signed; however, may be preceded by unsigned or signed (Except int64 may only be signed). Short and long may have the keyword INT following them with no effect. Also see #TYPE to change the default size.

SHORT INT1 is a special type used to generate very efficient code for bit operations and I/O. Arrays of bits (INT1 or SHORT) in RAM are now supported. Pointers to bits are not permitted. The device header files contain defines for BYTE as an int8 and BOOLEAN as an int1.

Integers are stored in little endian format. The LSB is in the lowest address. Float formats are described in common questions.

SEE ALSO: Declarations, Type Qualifiers, Enumerated Types, Structures & Unions, typedef, Named Registers

---

**Type Qualifiers**
<table>
<thead>
<tr>
<th>Type-Qualifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>static</strong></td>
<td>Variable is globally active and initialized to 0. Only accessible from this compilation unit.</td>
</tr>
<tr>
<td><strong>auto</strong></td>
<td>Variable exists only while the procedure is active. This is the default and AUTO need not be used.</td>
</tr>
<tr>
<td><strong>double</strong></td>
<td>Is a reserved word but is not a supported data type.</td>
</tr>
<tr>
<td><strong>extern</strong></td>
<td>External variable used with multiple compilation units. No storage is allocated. Is used to make otherwise out of scope data accessible. There must be a non-extern definition at the global level in some compilation unit.</td>
</tr>
<tr>
<td><strong>register</strong></td>
<td>Is allowed as a qualifier however, has no effect.</td>
</tr>
<tr>
<td><strong>_fixed(n)</strong></td>
<td>Creates a fixed point decimal number where ( n ) is how many decimal places to implement.</td>
</tr>
<tr>
<td><strong>unsigned</strong></td>
<td>Data is always positive. This is the default data type if not specified.</td>
</tr>
<tr>
<td><strong>signed</strong></td>
<td>Data can be negative or positive.</td>
</tr>
<tr>
<td><strong>volatile</strong></td>
<td>Tells the compiler optimizer that this variable can be changed at any point during execution.</td>
</tr>
<tr>
<td><strong>const</strong></td>
<td>Data is read-only. Depending on compiler configuration, this qualifier may just make the data read-only -AND/OR- it may place the data into program memory to save space. (see #DEVICE const=)</td>
</tr>
<tr>
<td><strong>rom</strong></td>
<td>Forces data into program memory. Pointers may be used to this data but they can not be mixed with RAM pointers.</td>
</tr>
<tr>
<td><strong>void</strong></td>
<td>Built-in basic type. Type void is used to indicate no specific type in places where a type is required.</td>
</tr>
<tr>
<td><strong>readonly</strong></td>
<td>Writes to this variable should be dis-allowed</td>
</tr>
<tr>
<td><strong>_bif</strong></td>
<td>Used for compiler built in function prototypes on the same line</td>
</tr>
<tr>
<td><strong><strong>attribute</strong></strong></td>
<td>Sets various attributes</td>
</tr>
</tbody>
</table>
Enumerated Types

**enum** enumeration type: creates a list of integer constants.

```
enum [id] { [ id [ = cexpr]] }
```

The id after **enum** is created as a type large enough to the largest constant in the list. The ids in the list are each created as a constant. By default the first id is set to zero and they increment by one. If *a = cexpr* follows an id that id will have the value of the constant expression and the following list will increment by one.

For example:
```
enum colors{red, green=2, blue};  // red will be 0,
green will be 2 and              // blue will be 3
```

Structures and Unions

**Struct** structure type: creates a collection of one or more variables, possibly of different types, grouped together as a single unit.

```
struct[+] [id] type-qualifier [+] id [:bits]; } [id]
```

For example:
```
struct data_record {
    int a[2];
```
Data Definitions

```c
int b : 2; /*2 bits */
int c : 3; /*3 bits*/
int d;
} data_var;       //data_record is a structure
type
        //data_var is a variable
```

**Union** type: holds objects of different types and sizes, with the compiler keeping track of size and alignment requirements. They provide a way to manipulate different kinds of data in a single area of storage.

<table>
<thead>
<tr>
<th>Union[<em>] [id] { type-qualifier [</em>] id [:bits]; } [id]</th>
</tr>
</thead>
<tbody>
<tr>
<td>One or more, semi-colon separated</td>
</tr>
<tr>
<td>Zero or more</td>
</tr>
</tbody>
</table>

For example:
```c
union u_tab {
    int ival;
    long lval;
    float fval;
};             //u_tag is a union type that can hold a float
```

**typedef**

If **typedef** is used with any of the basic or special types it creates a new type name that can be used in declarations. The identifier does not allocate space but rather may be used as a type specifier in other data definitions.

<table>
<thead>
<tr>
<th>typedef</th>
<th>[type-qualifier] [declarator];</th>
<th>[type-specifier]</th>
</tr>
</thead>
</table>

For example:
typedef int mybyte; // mybyte can be used in
// declaration to
int type

typedef short mybit; // mybyte can be used in
// declaration to
int type

typedef enum {red, green=2, blue}colors; // colors can be used to declare
// variable of this enum type

Non-RAM Data Definitions

CCS C compiler also provides a custom qualifier addressmod which can be used to define a memory region that can be RAM, program eeprom, data eeprom or external memory. Addressmod replaces the older typemod (with a different syntax).

The usage is :
addressmod
(name, read_function, write_function, start_address, end_address, share);

Where the read_function and write_function should be blank for RAM, or for other memory should be the following prototype:

// read procedure for reading n bytes from the memory starting at location addr
void read_function(int32 addr, int8 *ram, int nbytes){
}

// write procedure for writing n bytes to the memory starting at location addr
void write_function(int32 addr, int8 *ram, int nbytes) {
}

For RAM the share argument may be true if unused RAM in this area can be used by the compiler for standard variables.

Example:
void DataEE_Read(int32 addr, int8 *ram, int bytes) {
    int i;
    for (i=0;i<bytes;i++, ram++, addr++)
        *ram = read_eeprom(addr);
}

void DataEE_Write(int32 addr, int8 *ram, int bytes) {
    int i;
    for (i=0;i<bytes;i++, ram++, addr++)
        write_eeprom(addr,*ram);
}

addressmod
(DataEE, DataEE_read, DataEE_write, 5, 0xff);
    // would define a region called DataEE
    // between
    // 0x5 and 0xff in the chip data EEPROM.

void main (void) {
    int DataEE test;
    int x, y;
    x = 12;
    test = x; // writes x to the Data EEPROM
    y = test; // Reads the Data EEPROM
}

Note: If the area is defined in RAM then read and write functions are not required, the variables assigned in the memory region defined by the addressmod can be treated as a regular variable in all valid expressions. Any structure or data type can be used with an addressmod. Pointers can also be made to an addressmod data type. The #type directive can be used to make this memory region as default for variable allocations.

The syntax is:
Using Program Memory for Data

CCS C Compiler provides a few different ways to use program memory for data. The different ways are discussed below:

Constant Data:
The `const` qualifier will place the variables into program memory. If the keyword `const` is used before the identifier, the identifier is treated as a constant. Constants should be initialized and may not be changed at run-time. This is an easy way to create lookup tables.

The `rom` Qualifier puts data in program memory with 3 bytes per instruction space. The address used for ROM data is not a physical address but rather a true byte address. The & operator can be used on ROM variables however the address is logical not physical.

The syntax is:

```
const type id[cexpr] = {value}
```

For example:

Placing data into ROM
```
const int table[16]={0,1,2...15}
```

Placing a string into ROM
```
const char cstring[6]="hello"
```

Creating pointers to constants
```
const char *cptr;
cptr = string;
```

The `#org` preprocessor can be used to place the constant to specified address blocks.

For example:

The constant ID will be at 1C00.
```
#ORG 0x1C00, 0x1C0F
```
Data Definitions

```c
CONST CHAR ID[10] = {"123456789"};
```

*Note:* Some extra code will precede the 123456789.

The function `label_address` can be used to get the address of the constant. The constant variable can be accessed in the code. This is a great way of storing constant data in large programs. Variable length constant strings can be stored into program memory.

A special method allows the use of pointers to ROM. This method does not contain extra code at the start of the structure as does constant.

For example:

```c
char rom commands[] = {"put", "get", "status", "shutdown"};
```

ROML may be used instead of ROM if you only to use even memory locations.

The compiler allows a non-standard C feature to implement a constant array of variable length strings.

The syntax is:

```c
const char id[n] [*] = { "string", "string" ...};
```

Where `n` is optional and `id` is the table identifier.

For example:

```c
const char colors[n] [*] = {"Red", "Green", "Blue"};
```

#ROM directive:

Another method is to use `#rom` to assign data to program memory.

The syntax is:

```c
#rom address = {data, data, … , data}
```

For example:

Places 1,2,3,4 to ROM addresses starting at 0x1000

```c
#rom 0x1000 = {1, 2, 3, 4}
```

Places null terminated string in ROM

```c
#rom 0x1000 = {"hello"}
```

This method can only be used to initialize the program memory.

Built-in-Functions:

The compiler also provides built-in functions to place data in program memory, they are:

- `write_program_memory(address, dataptr, count);`
  - Writes `count` bytes of data from `dataptr` to `address` in program memory.
  - Every fourth byte of data will not be written, fill with 0x00.

Please refer to the help of these functions to get more details on their usage and limitations regarding erase procedures. These functions can be used only on chips that allow writes to program memory. The compiler uses the flash memory erase and write routines to implement the functionality.
The data placed in program memory using the methods listed above can be read from
with the following functions:

- \texttt{read_program_memory((address, dataptr, count))}
  - Reads count bytes from program memory at address to RAM at dataptr. Every fourth byte of data is read as 0x00
- \texttt{read_rom_memory((address, dataptr, count))}
  - Reads count bytes from program memory at the logical address to RAM at dataptr.

These functions can be used only on chips that allow reads from program memory. The compiler uses the flash memory read routines to implement the functionality.

**Named Registers**

The CCS C Compiler supports the new syntax for filing a variable at the location of a processor register. This syntax is being proposed as a C extension for embedded use. The same functionality is provided with the non-standard \#byte, \#word, \#bit and \#locate.

The syntax is:

\begin{verbatim}
register _name type id;
Or
register constant type id;
\end{verbatim}

name is a valid SFR name with an underscore before it.

Examples:

\begin{verbatim}
register _status int8 status_reg;
register _T1IF int8 timer_interrupt;
register 0x04 int16 file_select_register;
\end{verbatim}
Function Definition

The format of a function definition is as follows:

```
[qualifier] id ( [type-specifier id] ) { [stmt] }
```

- **Optional See Below**
- Zero or more comma separated.
- See Data Types
- Zero or more Semi-colon separated. See Statements.

The qualifiers for a function are as follows:
- VOID
- type-specifier
- #separate
- #inline
- #int..

When one of the above are used and the function has a prototype (forward declaration of the function before it is defined) you must include the qualifier on both the prototype and function definition.

A (non-standard) feature has been added to the compiler to help get around the problems created by the fact that pointers cannot be created to constant strings. A function that has one CHAR parameter will accept a constant string where it is called. The compiler will generate a loop that will call the function once for each character in the string.

**Example:**

```c
void lcd_putc(char c) {
  ...
}

cd_putc("Hi There.");
```
Overloaded Functions

Overloaded functions allow the user to have multiple functions with the same name, but they must accept different parameters.

Here is an example of function overloading: Two functions have the same name but differ in the types of parameters. The compiler determines which data type is being passed as a parameter and calls the proper function.

This function finds the square root of a long integer variable.

```c
long FindSquareRoot(long n){
}
```

This function finds the square root of a float variable.

```c
float FindSquareRoot(float n){
}
```

FindSquareRoot is now called. If variable is of long type, it will call the first FindSquareRoot() example. If variable is of float type, it will call the second FindSquareRoot() example.

```c
result=FindSquareRoot(variable);
```

Reference Parameters

The compiler has limited support for reference parameters. This increases the readability of code and the efficiency of some inline procedures. The following two procedures are the same. The one with reference parameters will be implemented with greater efficiency when it is inline.

```c
funct_a(int*x, int*y){
    /*Traditional*/
    if(*x!=5)
        *y=*x+3;
}
funct_a(&a,&b);
```

```c
funct_b(int&x, int&y){
    /*Reference params*/
```
Function Definition

```c
if(x!=5)
    y=x+3;
}
funct_b(a,b);
```

Default Parameters

Default parameters allows a function to have default values if nothing is passed to it when called.

```c
int mygetc(char *c, int n=100){
}
```

This function waits n milliseconds for a character over RS232. If a character is received, it saves it to the pointer c and returns TRUE. If there was a timeout it returns FALSE.

```c
//gets a char, waits 100ms for timeout
mygetc(&c);
//gets a char, waits 200ms for a timeout
mygetc(&c, 200);
```

Variable Argument Lists

The compiler supports a variable number of parameters. This works like the ANSI requirements except that it does not require at least one fixed parameter as ANSI does. The function can be passed any number of variables and any data types. The access functions are VA_START, VA_ARG, and VA_END. To view the number of arguments passed, the NARGS function can be used.

```c
/*
 stdClass.h holds the macros and va_list data type needed for variable number of parameters.
 */
#include <stdarg.h>
```

A function with variable number of parameters requires two things. First, it requires the ellipsis (...), which must be the last parameter of the function. The ellipsis represents the variable argument list. Second, it requires one more variable before the ellipsis (...).
Usually you will use this variable as a method for determining how many variables have been pushed onto the ellipsis.

Here is a function that calculates and returns the sum of all variables:

```c
int Sum(int count, ...) {
    // a pointer to the argument list
    va_list al;
    int x, sum=0;
    // start the argument list
    // count is the first variable before the ellipsis
    va_start(al, count);
    while(count--) {
        // get an int from the list
        x = var_arg(al, int);
        sum += x;
    }
    // stop using the list
    va_end(al);
    return(sum);
}
```

Some examples of using this new function:

```c
x=Sum(5, 10, 20, 30, 40, 50);
y=Sum(3, a, b, c);
```
**FUNCTIONAL OVERVIEW**

**I2C**

I2C™ is a popular two-wire communication protocol developed by Philips. Many PIC microcontrollers support hardware-based I2C™. CCS offers support for the hardware-based I2C™ and a software-based master I2C™ device. (For more information on the hardware-based I2C module, please consult the datasheet for your target device; not all PICs support I2C™.)

<table>
<thead>
<tr>
<th>Relevant Functions:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i2c_start()</td>
<td>Issues a start command when in the I2C master mode.</td>
</tr>
<tr>
<td>i2c_write(data)</td>
<td>Sends a single byte over the I2C interface.</td>
</tr>
<tr>
<td>i2c_read()</td>
<td>Reads a byte over the I2C interface.</td>
</tr>
<tr>
<td>i2c_stop()</td>
<td>Issues a stop command when in the I2C master mode.</td>
</tr>
<tr>
<td>i2c_poll()</td>
<td>Returns a TRUE if the hardware has received a byte in the buffer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relevant Preprocessor:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#USE I2C</td>
<td>Configures the compiler to support I2C™ to your specifications.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relevant Interrupts:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#INT_SSP</td>
<td>I2C or SPI activity</td>
</tr>
<tr>
<td>#INT_BUSCOL</td>
<td>Bus Collision</td>
</tr>
<tr>
<td>#INT_I2C</td>
<td>I2C Interrupt (Only on 14000)</td>
</tr>
<tr>
<td>#INT_BUSCOL2</td>
<td>Bus Collision (Only supported on some PIC18's)</td>
</tr>
<tr>
<td>#INT_SSP2</td>
<td>I2C or SPI activity (Only supported on some PIC18's)</td>
</tr>
<tr>
<td>#INT_mi2c</td>
<td>Interrupts on activity from the master I2C module</td>
</tr>
<tr>
<td>#INT_si2c</td>
<td>Interrupts on activity form the slave I2C module</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relevant Include Files:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None, all functions built-in</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relevant getenv() Parameters:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C_SLAVE</td>
<td>Returns a 1 if the device has I2C slave H/W</td>
</tr>
<tr>
<td>I2C_MASTER</td>
<td>Returns a 1 if the device has a I2C master H/W</td>
</tr>
</tbody>
</table>
Example Code:
#define Device_SDA PIN_C3      // Pin defines
#define Device_SCL PIN_C4

#include i2c(master,
sda=Device_SDA,
scl=Device_SCL)

.. 

BYTE data;         // Data to be transmitted
i2c_start();       // Issues a start command when in the I2C master mode.
i2c_write(data);   // Sends a single byte over the I2C interface.
i2c_stop();        // Issues a stop command when in the I2C master mode.

ADC

These options let the user configure and use the analog to digital converter module.
They are only available on devices with the ADC hardware. The options for the functions
and directives vary depending on the chip and are listed in the device header file. On
some devices there are two independent ADC modules, for these chips the second
module is configured using secondary ADC setup functions (Ex. setup_ADC2).

Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup_adc(mode)</td>
<td>Sets up the a/d mode like off, the adc clock etc.</td>
</tr>
<tr>
<td>setup_adc_ports(value)</td>
<td>Sets the available adc pins to be analog or digital.</td>
</tr>
<tr>
<td>set_adc_channel(channel)</td>
<td>Specifies the channel to be use for the a/d call.</td>
</tr>
<tr>
<td>read_adc(mode)</td>
<td>Starts the conversion and reads the value. The mode can also control the functionality.</td>
</tr>
<tr>
<td>adc_done()</td>
<td>Returns 1 if the ADC module has finished its conversion.</td>
</tr>
<tr>
<td>setup_adc2(mode)</td>
<td>Sets up the ADC2 module, for example the ADC clock and ADC sample time.</td>
</tr>
<tr>
<td>setup_adc_ports2(ports,</td>
<td>Sets the available ADC2 pins to be analog or digital, and</td>
</tr>
<tr>
<td>reference)</td>
<td>sets the voltage reference for ADC2.</td>
</tr>
<tr>
<td>set_adc_channel2(channel)</td>
<td>Specifies the channel to use for the ADC2 input.</td>
</tr>
<tr>
<td>read_adc2(mode)</td>
<td>Starts the sample and conversion sequence and reads the value. The mode can also control the functionality.</td>
</tr>
<tr>
<td>adc_done()</td>
<td>Returns 1 if the ADC module has finished its conversion.</td>
</tr>
</tbody>
</table>

Relevant Preprocessor:

<table>
<thead>
<tr>
<th>Preprocessor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#DEVICE ADC=xx</td>
<td>Configures the read_adc return size. For example, using a PIC with a 10 bit A/D you can use 8 or 10 for xx- 8 will return the most significant byte, 10 will return the full A/D</td>
</tr>
</tbody>
</table>
Functional Overview

Reading of 10 bits.

Relevant Interrupts:

- **INT_AD**: Interrupt fires when a/d conversion is complete
- **INT_ADOF**: Interrupt fires when a/d conversion has timed out

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

- **ADC_CHANNELS**: Number of A/D channels
- **ADC_RESOLUTION**: Number of bits returned by read_adc

Example Code:

```c
#define DEVICE ADC=10
...
long value;
...
ssetup_adc(ADC_CLOCK_INTERNAL);   //enables the a/d module
   //and sets the clock to internal adc clock
setup_adc_ports(ALL_ANALOG);     //sets all the adc pins to analog
set_adc_channel(0);              //the next read_adc call will read channel 0
delay_us(10);                    //a small delay is required after setting the channel
   //and before read
value=read_adc();                //starts the conversion and reads the result
   //and store it in value
read_adc(ADC_STARTONLY);         //only starts the conversion
value=read_adc(ADC_READONLY);    //reads the result of the last conversion and store it in
   //value. Assuming the device hat a 10bit ADC module,
   //value will range between 0-3FF. If #DEVICE ADC=8 had
   //been used instead the result will yield 0-FF. If #DEVICE
   //ADC=16 had been used instead the result will yield 0-
   //FFC0
```

Analog Comparator

These functions set up the analog comparator module. Only available in some devices.

Relevant Functions:
setup_comparator(mode) Enables and sets the analog comparator module. The options vary depending on the chip. Refer to the header file for details.

Relevant Preprocessor:
None

Relevant Interrupts:
INT_COMP Interrupt fires on comparator detect. Some chips have more than one comparator unit, and thus, more interrupts.

Relevant Include Files:
None, all functions built-in

Relevant getenv() Parameters:
Returns 1 if the device has a comparator COMP

Example Code:
setup_comparator(A4_A5_NC_NC);
if(C1OUT)
output_low(PIN_D0);
else
output_high(PIN_D1);

CAN Bus

These functions allow easy access to the Controller Area Network (CAN) features included with the MCP2515 CAN interface chip and the PIC24, dsPIC30 and dsPIC33 MCUs. These functions will only work with the MCP2515 CAN interface chip and PIC microcontroller units containing either a CAN or an ECAN module. Some functions are only available for the ECAN module and are specified by the word ECAN at the end of the description. The listed interrupts are not available to the MCP2515 interface chip.

Relevant Functions:
can_init(void); Initializes the module to 62.5k baud for ECAN and 125k baud for CAN and clears all the filters and masks so that all messages can be received from any ID.
## Functional Overview

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>can_set_baud(void);</code></td>
<td>Initializes the baud rate of the bus to 62.5kHz for ECAN and 125kHz for CAN. It is called inside the <code>can_init()</code> function so there is no need to call it.</td>
</tr>
<tr>
<td><code>can_set_mode (CAN_OP_MODE mode);</code></td>
<td>Allows the mode of the CAN module to be changed to listen all mode, configuration mode, listen mode, loop back mode, disabled mode, or normal mode.</td>
</tr>
<tr>
<td><code>can_set_functional_mode (CAN_FUN_OP_MODE mode);</code></td>
<td>Allows the functional mode of ECAN modules to be changed to legacy mode, enhanced legacy mode, or first in firstout (fifo) mode. ECAN</td>
</tr>
<tr>
<td><code>can_set_id(int16 *addr, int32 id, int1 ext)</code></td>
<td>Can be used to set the filter and mask ID's to the value specified by addr. It is also used to set the ID of the message to be sent on CAN chips.</td>
</tr>
<tr>
<td><code>can_set_buffer_id(BUFFER buffer, int32 id, int1 ext)</code></td>
<td>Can be used to set the ID of the message to be sent for ECAN chips. ECAN</td>
</tr>
<tr>
<td><code>can_get_id(BUFFER buffer, int1 ext)</code></td>
<td>Returns the ID of a received message.</td>
</tr>
<tr>
<td><code>can_putd(int32 id, int8 *data, int8 len, int8 priority, int1 ext, int1 rtr)</code></td>
<td>Constructs a CAN packet using the given arguments and places it in one of the available transmit buffers.</td>
</tr>
<tr>
<td><code>can_getd(int32 &amp;id, int8 *data, int8 &amp;len, struct rx_stat &amp;stat)</code></td>
<td>Retrieves a received message from one of the CAN buffers and stores the relevant data in the referenced function parameters.</td>
</tr>
<tr>
<td><code>can_kbhit()</code></td>
<td>Returns TRUE if valid CAN messages is available to be retrieved from one of the receive buffers.</td>
</tr>
<tr>
<td><code>can_tbe()</code></td>
<td>Returns TRUE if a transmit buffer is is available to send more data.</td>
</tr>
<tr>
<td><code>can_abort()</code></td>
<td>Aborts all pending transmissions.</td>
</tr>
<tr>
<td><code>can_enable_b_transfer(BUFFER b)</code></td>
<td>Sets the specified programmable buffer to be a transmit buffer. ECAN</td>
</tr>
<tr>
<td><code>can_enable_b_receiver(BUFFER b)</code></td>
<td>Sets the specified programmable buffer to be a receive buffer. By default all programmable buffers are set to be receive buffers. ECAN</td>
</tr>
<tr>
<td><code>can_enable_rtr(BUFFER b)</code></td>
<td>Enables the automatic response feature which automatically sends a user created packet when a specified ID is received. ECAN</td>
</tr>
<tr>
<td><code>can_disable_rtr(BUFFER b)</code></td>
<td>Disables the automatic response feature. ECAN</td>
</tr>
</tbody>
</table>
| `can_load_rtr (BUFFER b, int8)`             | Creates and loads the packet that will automatically
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>can_set_buffer_size(int8 size)</code></td>
<td>Set the number of buffers to use. Size can be 4, 6, 8, 12, 16, 24, and 32. By default <code>can_init()</code> sets size to 32. ECAN</td>
</tr>
<tr>
<td><code>can_enable_filter(CAN_FILTER_CONTROL filter)</code></td>
<td>Enables one of the acceptance filters included in the ECAN module. ECAN</td>
</tr>
<tr>
<td><code>can_disable_filter(CAN_FILTER_CONTROL filter)</code></td>
<td>Disables one of the acceptance filters included in the ECAN module. ECAN</td>
</tr>
<tr>
<td><code>can_associate_filter_to_buffer(CAN_FILTER_ASSOCIATION_BUFFERS buffer, CAN_FILTER_ASSOCIATION filter)</code></td>
<td>Used to associate a filter to a specific buffer. This allows only specific buffers to be filtered and is available in the ECAN module. ECAN</td>
</tr>
<tr>
<td><code>can_associate_filter_to_mask(CAN_MASK_FILTER_ASSOCIATION mask, CAN_FILTER_ASSOCIATION filter)</code></td>
<td>Used to associate a mask to a specific buffer. This allows only specific buffer to have this mask applied. This feature is available in the ECAN module. ECAN</td>
</tr>
<tr>
<td><code>can_fifo_getd(int32 &amp;id, int8 *data, int8 &amp;len, struct rx_stat &amp;stat )</code></td>
<td>Retrieves the next buffer in the FIFO buffer. Only available in the ECAN module. ECAN</td>
</tr>
<tr>
<td><code>can_trb0_putd(int32 id, int8 *data, int8 len, int8 pri, int1 ext, int1 rtr)</code></td>
<td>Constructs a CAN packet using the given arguments and places it in transmit buffer 0. Similar functions available for all transmit buffers 0-7. Buffer must be made a transmit buffer with <code>can_enable_b_transfer()</code> function before function can be used. ECAN</td>
</tr>
</tbody>
</table>
| `can_enable_interruptions(INTERRUPT setting)` | Enables specified interrupt conditions that cause the #INT_CAN1 interrupt to be triggered. Available options are: TB - Transmitt Buffer Interrupt
RX - Receive Buffer Interrupt
RXOV - Receive Buffer Overflow Interrupt
FIFO - FIFO Almost Full Interrupt
ERR - Error interrupt
WAK - Wake-Up Interrupt
IVR - Invalid Message Received Interrupt
RX0 - Receive Buffer 0 Interrupt
RX1 - Receive Buffer 1 Interrupt
TX0 - Transmit Buffer 0 Interrupt
TX1 - Transmit Buffer 1 Interrupt
TX2 - Transmit Buffer 2 Interrupt
CAN |
| `can_disable_interruptions(INTERRUPT setting)` | Disable specified interrupt conditions so they doesn't cause |
### Functional Overview

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RRUPT setting)</strong></td>
<td>the #INT_CAN1 interrupt to be triggered. Available options are the same as for the <code>can_enable_interrupts()</code> function. By default all conditions are disabled.</td>
</tr>
<tr>
<td><strong>can_config_DMA(void)</strong></td>
<td>Configures the DMA buffers to use with the ECAN module. It is called inside the <code>can_init()</code> function so there is no need to call it. ECAN</td>
</tr>
<tr>
<td><strong>For PICs that have two CAN or ECAN modules all the above function are available for the second module, and they start with can2 instead of can.</strong></td>
<td>Examples: can2_init(); can2_kbhit();</td>
</tr>
<tr>
<td><strong>Relevant Preprocessor:</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Relevant Interrupts:</strong></td>
<td></td>
</tr>
<tr>
<td>#INT_CAN1</td>
<td>Interrupt for CAN or ECAN module 1. This interrupt is triggered when one of the conditions set by the <code>can_enable_interrupts()</code> is meet.</td>
</tr>
<tr>
<td>#INT_CAN2</td>
<td>Interrupt for CAN or ECAN module 2. This interrupt is triggered when one of the conditions set by the <code>can2_enable_interrupts()</code> is meet. This interrupt is only available on PICs that have two CAN or ECAN modules.</td>
</tr>
<tr>
<td><strong>Relevant Include Files:</strong></td>
<td></td>
</tr>
<tr>
<td>can-mcp2510.c</td>
<td>Drivers for the MCP2510 and MCP2515 interface chips.</td>
</tr>
<tr>
<td>can-dsPIC30.c</td>
<td>Drivers for the built in CAN module on dsPIC30F chips.</td>
</tr>
<tr>
<td>can-PIC24.c</td>
<td>Drivers for the build in ECAN module on PIC24HJ and dsPIC33FJ chips.</td>
</tr>
<tr>
<td><strong>Relevant getenv() Parameters:</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Example Code:</strong></td>
<td></td>
</tr>
<tr>
<td><code>can_init();</code></td>
<td>// Initializes the CAN bus.</td>
</tr>
<tr>
<td><code>can_putd(0x300,data,8,3,TRUE,FALSE);</code></td>
<td>// Places a message on the CAN bus with</td>
</tr>
<tr>
<td></td>
<td>// “data”, the TRUE causes an extended ID to be</td>
</tr>
<tr>
<td></td>
<td>// sent, the FALSE causes no remote transmission</td>
</tr>
<tr>
<td></td>
<td>// to be requested.</td>
</tr>
<tr>
<td><code>can_getd(ID,data,len,stat);</code></td>
<td>// Retrieves a message from the CAN bus storing the</td>
</tr>
<tr>
<td></td>
<td>// ID in the ID variable, the data at the array</td>
</tr>
<tr>
<td></td>
<td>// pointed to by</td>
</tr>
<tr>
<td></td>
<td>// to by “data”, the number of data bytes in len and</td>
</tr>
</tbody>
</table>
Code Profile

Profile a program while it is running. Unlike in-circuit debugging, this tool grabs information while the program is running and provides statistics, logging and tracing of it's execution. This is accomplished by using a simple communication method between the processor and the ICD with minimal side-effects to the timing and execution of the program. Another benefit of code profile versus in-circuit debugging is that a program written with profile support enabled will run correctly even if there is no ICD connected.

In order to use Code Profiling, several functions and pre-processor statements need to be included in the project being compiled and profiled. Doing this adds the proper code profile run-time support on the microcontroller.

See the help file in the Code Profile tool for more help and usage examples.

Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>profileout()</code></td>
<td>Send a user specified message or variable to be displayed or logged by the code profile tool.</td>
</tr>
</tbody>
</table>

Relevant Pre-Processor:

<table>
<thead>
<tr>
<th>Pre-Processor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#use profile()</code></td>
<td>Global configuration of the code profile run-time on the microcontroller.</td>
</tr>
<tr>
<td><code>#profile</code></td>
<td>Dynamically enable/disable specific elements of the profiler.</td>
</tr>
</tbody>
</table>

Relevant Interrupts:

<table>
<thead>
<tr>
<th>Interrupts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The profiler can be configured to use a microcontroller's internal timer for more accurate timing of events over the clock on the PC. This timer is configured using the <code>#profile</code> pre-processor command.</td>
</tr>
</tbody>
</table>
### Functional Overview

**Relevant Include Files:** None – all the functions are built into the compiler.

**Relevant getenv():** None

**Example Code:**
```
#include <18F4520.h>
#define delay(crystal=10MHz, clock=40MHz)
#define profile functions, parameters
void main(void)
{
    int adc;
    setup_adc(ADC_CLOCK_INTERNAL);
    set_adc_channel(0);

    for(;;)
    {
        adc = read_adc();
        profileout(adc);
        delay_ms(250);
    }
}
```

### Configuration Memory

On all dsPIC30, dsPIC33 and PIC24 families, the configuration memory is readable and writable. The configuration memory contains the configuration bits for things such as the oscillator mode, watchdog timer enable, etc. These configuration bits are set by the CCS C compiler usually through a #fuse. CCS provides an API that allows these bits to be changed in run-time.

**Relevant Functions:**

- `write_configuration_memory(ramPtr, n);` Writes n bytes to configuration from ramPtr, no erase needed

or

- `write_configuration_memory` Write n bytes to configuration memory, starting at offset,
(offset, ramPtr, n); from ramPtr */

read_configuration_memory (ramPtr, n); Read n bytes of configuration memory, save to ramPtr

Relevant Preprocessor: The initial value of the configuration memory is set through a #FUSE

Relevant Interrupts: None

Relevant Include Files: None, all functions built-in

Relevant getenv() parameters: None

Example Code:

```c
int16 data = 0x0C32;
write_configuration_memory (&data, 2); //writes 2 bytes to the configuration memory
```

## CRC

The programmable Cyclic Redundancy Check (CRC) is a software configurable CRC checksum generator in select PIC24F, PIC24H, PIC24EP, and dsPIC33EP devices. The checksum is a unique number associated with a message or a block of data containing several bytes. The built-in CRC module has the following features:

- Programmable bit length for the CRC generator polynomial. (up to 32 bit length)
- Programmable CRC generator polynomial.
- Interrupt output.
- 4-deep, 8-deep, 16-bit, 16-deep or 32-deep, 8-bit FIFO for data input.
- Programmed bit length for data input. (32-bit CRC Modules Only)

Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup_crc (polynomial)</td>
<td>This will setup the CRC polynomial.</td>
</tr>
<tr>
<td>crc_init (data)</td>
<td>Sets the initial value used by the CRC module.</td>
</tr>
<tr>
<td>crc_calc (data)</td>
<td>Returns the calculated CRC value.</td>
</tr>
</tbody>
</table>

Relevant Preprocessor: None

Relevant Interrupts: None
#INT_CRC

On completion of CRC calculation.

Relevant Include Files:

**None, all functions built-in**

Relevant getenv() parameters:

**None**

Example Code:

```c
Int16 data[8];
int16 result;

setup_crc(15, 3, 1);  // CRC Polynomial is X16 + X15 + X3 + X1+ 1 or Polynomial = 8005h

crc_init(0xFEEE);  // Starts the CRC accumulator out at 0xFEEE

Result = crc_calc(&data[0], 8);  // Calculate the CRC
```

## DAC

These options let the user configure and use the digital to analog converter module. They are only available on devices with the DAC hardware. The options for the functions and directives vary depending on the chip and are listed in the device header file.

Relevant Functions:

- **setup_dac(divisor)**  
  Sets up the DAC e.g. Reference voltages
- **dac_write(value)**  
  Writes the 8-bit value to the DAC module
- **setup_dac(mode, divisor)**  
  Sets up the d/a mode e.g. Right enable, clock divisor
- **dac_write(channel, value)**  
  Writes the 16-bit value to the specified channel

Relevant Preprocessor:

- **#USE DELAY**  
  Must add an auxiliary clock in the #use delay preprocessor. For example:
  
  ```c
  #USE DELAY(clock=20M, Aux: crystal=6M, clock=3M)
  ```

Relevant Interrupts:  

**None**

Relevant Include Files:

**None, all functions built-in**

Relevant getenv() parameters:

**None**

Example Code:

```c
int16 i = 0;
```
setup_dac(DAC_RIGHT_ON, 5);  // enables the d/a module with right channel
a division of the clock by 5
While(1){
i++;
dac_write(DAC_RIGHT, i);  // writes i to the right DAC channel
}

Data EEPROM

The data EEPROM memory is readable and writable in some chips. These options let the user read and write to the data EEPROM memory. These functions are only available in flash chips.

Relevant Functions:

- **read_eeprom(address)**: Reads the data EEPROM memory location
- **write_eeprom(address, value)**: Erases and writes value to data EEPROM location address.
- **read_eeprom(address, [N])**: Reads N bytes of data EEPROM starting at memory location address. The maximum return size is int64.
- **read_eeprom(address, [variable])**: Reads from EEPROM to fill variable starting at address
- **read_eeprom(address, pointer, N)**: Reads N bytes, starting at address, to pointer
- **write_eeprom(address, value)**: Writes value to EEPROM address
- **write_eeprom(address, pointer, N)**: Writes N bytes to address from pointer

Relevant Preprocessor:

```
#ROM address={list}
```

Can also be used to put data EEPROM memory data into the hex file.
**Functional Overview**

**write_eeprom = noint**

Allows interrupts to occur while the `write_eeprom()` operations is polling the done bit to check if the write operations has completed. Can be used as long as no EEPROM operations are performed during an ISR.

Relevant Interrupts:

**INT_EEPROM**

Interrupt fires when EEPROM write is complete

Relevant Include Files:

**None, all functions built-in**

Relevant getenv() parameters:

**DATA_EEPROM**

Size of data EEPROM memory.

Example Code:

```c
#define DATA_EEPROM 0x007FFC00 /* Inserts this data into the hex file */
#define RX_CONFIG {1,2,3,4,5} /* The data EEPROM address differs between PICs */
#define TX_CONFIG 0x1337 /* Please refer to the device editor for device specific values. */

void write_eeprom(uint16_t address, uint16_t data) {
    // Writes 0x1337 to data EEPROM location 10.
    value = read_eeprom(address);
    // Reads data EEPROM location 10 returns 0x1337.
}
```

**DCI**

DCI is an interface that is found on several dsPIC devices in the 30F and the 33FJ families. It is a multiple-protocol interface peripheral that allows the user to connect to many common audio codecs through common (and highly configurable) pulse code modulation transmission protocols. Generic multichannel protocols, I2S and AC’97 (16 & 20 bit modes) are all supported.

Relevant Functions:

- `setup_dci(configuration, data size, rx config, tx config, sample rate);`
  Initializes the DCI module.
- `setup_adc_ports(value)`
  Sets the available adc pins to be analog or digital.
- `set_adc_channel(channel)`
  Specifies the channel to be use for the a/d call.
- `read_adc(mode)`
  Starts the conversion and reads the value. The mode can also control the functionality.
- `adc_done()`
  Returns 1 if the ADC module has finished its conversion.
Relevant Preprocessor:

| #DEVICE ADC=xx | Configures the read_adc return size. For example, using a PIC with a 10 bit A/D you can use 8 or 10 for xx- 8 will return the most significant byte, 10 will return the full A/D reading of 10 bits. |

Relevant Interrupts:

| INT_DCI | Interrupt fires on a number (user configurable) of data words received. |

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

None

Example Code:

```c
signed int16 left_channel, right_channel;

dci_initialize((I2S_MODE | DCI_MASTER | DCI_CLOCK_OUTPUT |
SAMPLE_RISING_EDGE | UNDERFLOW_LAST |
MULTI_DEVICE_BUS),DCI_1WORD_FRAME |
DCI_16BIT_WORD | DCI_2WORD_INTERRUPT, RECEIVE_SLOT0 |
RECEIVE_SLOT1, 
TRANSMIT_SLOT0 | TRANSMIT_SLOT1, 6000);

...

dci_start();

...

while(1)
{
    dci_read(&left_channel, &right_channel);
    dci_write(&left_channel, &right_channel);
}
```

DMA

The Direct Memory Access (DMA) controller facilitates the transfer of data between the CPU and its peripherals without the CPU's assistance. The transfer takes place between peripheral data registers and data space RAM. The module has 8 channels and since
Functional Overview

Each channel is unidirectional, two channels must be allocated to read and write to a peripheral. Each DMA channel can move a block of up to 1024 data elements after it generates an interrupt to the CPU to indicate that the lock is available for processing. Some of the key features of the DMA module are:

- Eight DMA Channels.
- Byte or word transfers.
- CPU interrupt after half or full block transfer complete.
- One-Shot or Auto-Repeat block transfer modes.
- Ping-Pong Mode (automatic switch between two DSPRAM start addresses after each block transfer is complete).

<table>
<thead>
<tr>
<th>Relevant Functions:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>setup_dma(channel, peripheral, mode)</strong></td>
</tr>
<tr>
<td><strong>dma_start(channel, mode, address)</strong></td>
</tr>
<tr>
<td><strong>dma_status(channel)</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relevant Preprocessor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relevant Interrupts:</th>
</tr>
</thead>
<tbody>
<tr>
<td>#INT_DMAX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relevant Include Files:</th>
</tr>
</thead>
<tbody>
<tr>
<td>None, all functions built-in</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relevant getenv() parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

Example Code:

```c
setup_dma(1, DMA_IN_SPI1, DMA_BYTE);
```

Setup channel 1 of the DMA module to read the SPI1 channel in byte mode.

```c
dma_start(1, DMA_CONTINOUS| DMA_PING_PONG, 0x2000);
```

Start the DMA channel with the DMA RAM address of 0x2000.
**Data Signal Modulator**

The Data Signal Modulator (DSM) allows the user to mix a digital data stream (the “modulator signal”) with a carrier signal to produce a modulated output. Both the carrier and the modulator signals are supplied to the DSM module, either internally from the output of a peripheral, or externally through an input pin. The modulated output signal is generated by performing a logical AND operation of both the carrier and modulator signals and then it is provided to the MDOUT pin. Using this method, the DSM can generate the following types of key modulation schemes:

- Frequency Shift Keying (FSK)
- Phase Shift Keying (PSK)
- On-Off Keying (OOK)

**Relevant Functions:** *(8 bit or 16 bit depending on the device)*

- `setup_dsm(mode, source, carrier)`: Configures the DSM module and selects the source signal and carrier signals.
- `setup_dsm(TRUE)`: Enables the DSM module.
- `setup_dsm(FALSE)`: Disables the DSM module.

**Relevant Preprocessor:** None

**Relevant Interrupts:** None

**Relevant Include Files:** None, all functions built-in

**Relevant getenv() parameters:** None

**Example Code:**

```c
setup_dsm(DSM_ENABLED | DSM_OUTPUT_ENABLED, //Enables DSM module with the output enabled and selects UART1
          DSM_SOURCE_UART1, //as the source signal and VSS as the high carrier signal and OC1’s
          PWM output as the low carrier signal.
```
Extended RAM

Some PIC24 devices have more than 30K of RAM. For these devices a special method is required to access the RAM above 30K. This extended RAM is organized into pages of 32K bytes each, the first page of extended RAM starts on page 1.

Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>write_extended_ram(p, addr, ptr, n);</code></td>
<td>Writes n bytes from ptr to extended RAM page p starting at address addr.</td>
</tr>
<tr>
<td><code>read_extended_ram(p, addr, ptr, n);</code></td>
<td>Reads n bytes from extended RAM page p starting a address addr to ptr.</td>
</tr>
</tbody>
</table>

Relevant Preprocessor:
None

Relevant Interrupts:
None

Relevant Include Files:
None, all functions built-in

Relevant `getenv()` parameters:
None

Example Code:

```c
write_extended_ram(1,0x100,WriteData,8); // Writes 8 bytes from WriteData to addresses 0x100 to 0x107 of extended RAM page 1.
read_extended_ram(1,0x100,ReadData,8);  // Reads 8 bytes from addresses 0x100 to 0x107 of extended RAM page 1 to ReadData.
```
General Purpose I/O

These options let the user configure and use the I/O pins on the device. These functions will affect the pins that are listed in the device header file.

Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_high(pin)</td>
<td>Sets the given pin to high state.</td>
</tr>
<tr>
<td>output_low(pin)</td>
<td>Sets the given pin to the ground state.</td>
</tr>
<tr>
<td>output_float(pin)</td>
<td>Sets the specified pin to the input mode. This will allow the pin to float high to represent a high on an open collector type of connection.</td>
</tr>
<tr>
<td>output_x(value)</td>
<td>Outputs an entire byte to the port.</td>
</tr>
<tr>
<td>output_bit(pin,value)</td>
<td>Outputs the specified value (0,1) to the specified I/O pin.</td>
</tr>
<tr>
<td>input(pin)</td>
<td>The function returns the state of the indicated pin.</td>
</tr>
<tr>
<td>input_state(pin)</td>
<td>This function reads the level of a pin without changing the direction of the pin as INPUT() does.</td>
</tr>
<tr>
<td>set_tris_x(value)</td>
<td>Sets the value of the I/O port direction register. A '1' is an input and '0' is for output.</td>
</tr>
<tr>
<td>input_change_x()</td>
<td>This function reads the levels of the pins on the port, and compares them to the last time they were read to see if there was a change, 1 if there was, 0 if there was not.</td>
</tr>
<tr>
<td>set_open_drain_x(value)</td>
<td>This function sets the value of the I/O port Open-Drain register. A</td>
</tr>
<tr>
<td>set_input_level_x(value)</td>
<td>This function sets the value of the I/O port Input Level Register. A 1 sets the input level to ST and 0 sets the input level to TTL.</td>
</tr>
<tr>
<td>set_open_drain_x()</td>
<td>Sets the value of the I/O port Open-Drain Control register. A '1' sets it as an open-drain output, and a '0' sets it as a digital output.</td>
</tr>
</tbody>
</table>

Relevant Preprocessor:

#USE STANDARD_IO(port) This compiler will use this directive by default and it will automatically inserts code for the direction register whenever an I/O function like output_high() or input() is used.

#USE FAST_IO(port) This directive will configure the I/O port to use the fast method of performing I/O. The user will be responsible for setting the port direction register using the set_tris_x() function.
#USE FIXED_IO
(port_outputs=;in,pin?)

This directive sets particular pins to be used as an input or output, and the compiler will perform this setup every time this pin is used.

Relevant Interrupts: None

Relevant Include Files: None, all functions built-in

Relevant getenv() parameters: PIN:pb ---- Returns a 1 if bit b on port p is on this part

Example Code:

```
#use fast_io(b)
...
Int8 Tris_value= 0x0F;
int1 Pin_value;
set_tris_b(Tris_value); //Sets B0:B3 as input and B4:B7 as output
output_high(PIN_B7);   //Set the pin B7 to High
If(input(PIN_B0)){     //Read the value on
  pin B0, set B7 to low if pin B0 is high
  output_high(PIN_B7);}
```

## Input Capture

These functions allow for the configuration of the input capture module. The timer source for the input capture operation can be set to either Timer 2 or Timer 3. In capture mode the value of the selected timer is copied to the ICxBUF register when an input event occurs and interrupts can be configured to fire as needed.

Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup_capture(x, mode)</td>
<td>Sets the operation mode of the input capture module x</td>
</tr>
<tr>
<td>get_capture(x, wait)</td>
<td>Reads the capture event time from the ICxBUF result register. If wait is true, program flow waits until a new result is present. Otherwise the oldest value in the buffer is returned.</td>
</tr>
</tbody>
</table>

Relevant Preprocessor: None

Relevant Interrupts:
**Internal Oscillator**

Two internal oscillators are present in PCD compatible chips, a fast RC and slow RC oscillator circuit. In many cases (consult your target datasheet or family data sheet for target specifics) the fast RC oscillator may be connected to a PLL system, allowing a broad range of frequencies to be selected. The Watchdog timer is derived from the slow internal oscillator.

**Relevant Functions:**

| setup_oscillator() | Explicitly configures the oscillator. |

**Relevant Preprocessor:**

| \#FUSES | Specifies the values loaded in the device configuration memory. |

**Relevant Interrupts:**

| \#int_oscfail | Interrupts on oscillator failure |

**Relevant Include Files:**

| None, all functions built-in |

**Relevant getenv() parameters:**

| CLOCK | Returns the clock speed specified by \#use delay() |
| FUSE_SETxxxx | Returns 1 if the fuse xxxx is set. |

**Example Code:**

```c
setup_timer3(TMR_INTERNAL | TMR_DIV_BY_8);
setup_capture(2, CAPTURE_FE | CAPTURE_TIMER3);
while(TRUE) {
    timerValue = get_capture(2, TRUE);
    printf("A module 2 capture event occurred at: %LU", timerValue);
}
```
# Interrupts

The following functions allow for the control of the interrupt subsystem of the microcontroller. With these functions, interrupts can be enabled, disabled, and cleared. With the preprocessor directives, a default function can be called for any interrupt that does not have an associated ISR, and a global function can replace the compiler generated interrupt dispatcher.

## Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable_interrupts()</td>
<td>Disables the specified interrupt.</td>
</tr>
<tr>
<td>enable_interrupts()</td>
<td>Enables the specified interrupt.</td>
</tr>
<tr>
<td>ext_int_edge()</td>
<td>Enables the edge on which the edge interrupt should trigger. This can be either rising or falling edge.</td>
</tr>
<tr>
<td>clear_interrupt()</td>
<td>This function will clear the specified interrupt flag. This can be used if a global isr is used, or to prevent an interrupt from being serviced.</td>
</tr>
<tr>
<td>interrupt_active()</td>
<td>This function checks the interrupt flag of specified interrupt and returns true if flag is set.</td>
</tr>
<tr>
<td>interrupt_enabled()</td>
<td>This function checks the interrupt enable flag of the specified interrupt and returns TRUE if set.</td>
</tr>
</tbody>
</table>

## Relevant Preprocessor:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#INT_XXX level=x</td>
<td>x is an int 0-7, that selects the interrupt priority level for that interrupt.</td>
</tr>
<tr>
<td>#INT_XXX fast</td>
<td>This directive makes use of shadow registers for fast register save.</td>
</tr>
</tbody>
</table>

## Relevant Interrupts:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#int_default</td>
<td>This directive specifies that the following function should be called if an interrupt is triggered but no routine is associated with that interrupt.</td>
</tr>
<tr>
<td>#int_global</td>
<td>This directive specifies that the following function should</td>
</tr>
</tbody>
</table>
be called whenever an interrupt is triggered. This function will replace the compiler generated interrupt dispatcher.

`#int_xxx`  
This directive specifies that the following function should be called whenever the xxx interrupt is triggered. If the compiler generated interrupt dispatcher is used, the compiler will take care of clearing the interrupt flag bits.

Relevant Include Files:  
*none*, all functions built in.

Relevant `getenv()`  
Parameters:  
*none*

Example Code:
```
#int_timer0
void timer0interrupt() // #int_timer associates the following function with the
// interrupt service routine that should be called
enable_interrupts(TIMER0); // enables the timer0 interrupt
disable_interrupts(TIMER0); // disables the timer0 interrupt
clear_interrupt(TIMER0); // clears the timer0 interrupt flag
```

**Output Compare/PWM Overview**

The following functions are used to configure the output compare module. The output compare has three modes of functioning. Single compare, dual compare, and PWM. In single compare the output compare module simply compares the value of the OCxR register to the value of the timer and triggers a corresponding output event on match. In dual compare mode, the pin is set high on OCxR match and then placed low on an OCxRS match. This can be set to either occur once or repeatedly. In PWM mode the selected timer sets the period and the OCxRS register sets the duty cycle. Once the OC module is placed in PWM mode the OCxR register becomes read only so the value needs to be set before placing the output compare module in PWM mode. For all three modes of operation, the selected timer can either be Timer 2 or Timer 3.

Relevant Functions:

```c
setup_comparex (x, mode)  
Sets the mode of the output compare / PWM module x
```
### Functional Overview

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_comparex_time(x, ocr, [ocrs])</code></td>
<td>Sets the OCR and optionally OCRS register values of module x.</td>
</tr>
<tr>
<td><code>set_pwm_duty(x, value)</code></td>
<td>Sets the PWM duty cycle of module x to the specified value</td>
</tr>
</tbody>
</table>

**Relevant Preprocessor:**

None

**Relevant Interrupts:**

INT_OCx

Interrupt fires after a compare event has occurred

**Relevant Include Files:**

None, all functions built-in.

**Relevant getenv() parameters:**

None

**Example Code:**

```c
// Outputs a 1 second pulse on the OC2 PIN
// using dual compare mode on a PIC
// with an instruction clock of (20Mhz/4)
int16 OCR_2 = 0x1000;   // Start pulse when timer is at 0x1000
int16 OCRS_2 = 0x5C4B;  // End pulse after 0x04C4B timer counts (0x1000 + 0x04C4B)
    // (1 sec)/[(4/20000000)*256] = 0x04C4B
    // 256 = timer prescaler value (set in code below)
set_compare_time(2, OCR_2, OCRS_2);
setup_compare(2, COMPARE_SINGLE_PULSE | COMPARE_TIMER3);
setup_timer3(TMR_INTERNAL | TMR_DIV_BY_256);
```

## Motor Control PWM

These options lets the user configure the Motor Control Pulse Width Modulator (MCPWM) module. The MCPWM is used to generate a periodic pulse waveform which is useful in motor control and power control applications. The options for these functions vary depending on the chip and are listed in the device header file.

**Relevant Functions:**

`setup_motor_pwm(pwm,opt)` Configures the motor control PWM module.
ions, timebase);

**set_motor_pwm_duty(pwm, unit, time)**
Configures the motor control PWM unit duty.

**set_motor_pwm_event(pwm, time)**
Configures the PWM event on the motor control unit.

**set_motor_unit(pwm, unit, options, active_deadtime, inactive_deadtime);**
Configures the motor control PWM unit.

**get_motor_pwm_event(pwm);**
Returns the PWM event on the motor control unit.

Relevant Preprocessor:
None

Relevant Interrupts:
#INT_PWM1
PWM Timebase Interrupt

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:
None

Example Code:

```c
// Sets up the motor PWM module
setup_motor_pwm(1, MPWM_FREE_RUN | MPWM_SYNC_OVERRIDES, timebase);

// Sets the PWM1, Group 1 duty cycle value to 0x55
set_motor_pwm_duty(1, 1, 0x55);

// Set the motor PWM event
set_motor_pwm_event(pwm, time);

// Enable pwm pair
set_motor_unit(1, 1, mpwm_ENABLE, 0, 0);  // Enables pwm1, Group 1 in complementary mode, no deadtime
```
## PMP/EPMP

The Parallel Master Port (PMP)/Enhanced Parallel Master Port (EPMP) is a parallel 8-bit/16-bit I/O module specifically designed to communicate with a wide variety of parallel devices. Key features of the PMP module are:

- 8 or 16 Data lines
- Up to 16 or 32 Programmable Address Lines
- Up to 2 Chip Select Lines
- Programmable Strobe option
- Address Auto-Increment/Auto-Decrement
- Programmable Address/Data Multiplexing
- Programmable Polarity on Control Signals
- Legacy Parallel Slave (PSP) Support
- Enhanced Parallel Slave Port Support
- Programmable Wait States

### Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>setup_pmp</strong>(options,address_mask)</td>
<td>This will setup the PMP/EPMP module for various mode and specifies which address lines to be used.</td>
</tr>
<tr>
<td><strong>setup_psp</strong>(options,address_mask)</td>
<td>This will setup the PSP module for various mode and specifies which address lines to be used.</td>
</tr>
<tr>
<td><strong>setup_pmp_csx</strong>(options,[offset])</td>
<td>Sets up the Chip Select X Configuration, Mode and Base Address registers</td>
</tr>
<tr>
<td><strong>setup_psp_es</strong>(options)</td>
<td>Sets up the Chip Select X Configuration and Mode registers</td>
</tr>
<tr>
<td><strong>pmp_write</strong>(data)</td>
<td>Write the data byte to the next buffer location.</td>
</tr>
<tr>
<td><strong>psp_write</strong>(address,data) / <strong>psp_write</strong>(data)</td>
<td>This will write a byte of data to the next buffer location or will write a byte to the specified buffer location.</td>
</tr>
<tr>
<td><strong>pmp_read</strong>( )</td>
<td>Reads a byte of data.</td>
</tr>
<tr>
<td><strong>psp_read</strong>(address) / <strong>psp_read</strong>( )</td>
<td><strong>psp_read</strong>( ) will read a byte of data from the next buffer location and <strong>psp_read</strong>( address ) will read the buffer location address.</td>
</tr>
<tr>
<td><strong>pmp_address</strong>(address)</td>
<td>Configures the address register of the PMP module with the destination address during Master mode operation.</td>
</tr>
<tr>
<td><strong>pmp_overflow</strong>( )</td>
<td>This will return the status of the output buffer underflow bit.</td>
</tr>
<tr>
<td><strong>pmp_input_full</strong>( )</td>
<td>This will return the status of the input buffers.</td>
</tr>
<tr>
<td><strong>psp_input_full</strong>( )</td>
<td>This will return the status of the input buffers.</td>
</tr>
<tr>
<td><strong>pmp_output_full</strong>( )</td>
<td>This will return the status of the output buffers.</td>
</tr>
<tr>
<td><strong>psp_output_full</strong>( )</td>
<td>This will return the status of the output buffers.</td>
</tr>
</tbody>
</table>

### Relevant Preprocessor:

None
Relevant Interrupts:

#INT_PMP  
Interrupt on read or write strobe

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:
None

Example Code:
```
setup_pmp( PAR_ENABLE |                   // Sets up
          Master mode with
          PAR_MASTER_MODE_1|   // address
          PAR_STOP_IN_IDLE,0x0FF);

if (pmp_output_full())
{
    pmp_write(next_byte);
}
```

---

**Program Eeprom**

The Flash program memory is readable and writable in some chips and is just readable in some. These options lets the user read and write to the Flash program memory. These functions are only available in flash chips.

Relevant Functions:

- **read_program_eeprom(address)**  
  Reads the program memory location (16 bit or 32 bit depending on the device).

- **write_program_eeprom(address, value)**  
  Writes value to program memory location address.

- **erase_program_eeprom(address)**  
  Erases FLASH_ERASE_SIZE bytes in program memory.

- **write_program_memory(address, dataptr, count)**  
  Writes count bytes to program memory from dataptr to address. When address is a multiple of FLASH_ERASE_SIZE an erase is also performed.

- **read_program_memory(address, count)**  
  Read count bytes from program memory at address to
### Functional Overview

**write_rom_memory**

(address, dataptr, count)

Writes `count` bytes to program memory from `address` (32 bits)

**read_rom_memory**

(address, dataptr, count)

Read `count` bytes to program memory from `address` (32 bits)

**Relevant Preprocessor:**

- **#ROM address={list}**
  
  Can be used to put program memory data into the hex file.

- **#DEVICE(WRITE_EEPROM=ASYNC)**
  
  Can be used with #DEVICE to prevent the write function from hanging. When this is used make sure the eeprom is not written both inside and outside the ISR.

**Relevant Interrupts:**

- **INT_EEPROM**
  
  Interrupt fires when eeprom write is complete.

**Relevant Include Files:**

None, all functions built-in

**Relevant getenv() parameters**

- **PROGRAM_MEMORY**
  
  Size of program memory

- **READ_PROGRAM**
  
  Returns 1 if program memory can be read

- **FLASH_WRITE_SIZE**
  
  Smallest number of bytes written in flash

- **FLASH_ERASE_SIZE**
  
  Smallest number of bytes erased in flash

**Example Code:**

```c
#ROM
0x300={1,2,3,4} // Inserts this data into the hex file.

erase_program_eeprom(0x00000300); // Erases 32 instruction locations starting at 0x0300

write_program_eeprom(0x00000300,0x123456); // Writes 0x123456 to 0x0300

value=read_program_eeprom(0x0000300); // Reads 0x0300 returns 0x123456

write_program_memory(0x00000300, data, 12); // Erases 32 instructions starting
```
The Quadrature Encoder Interface (QEI) module provides the interface to incremental encoders for obtaining mechanical positional data.
### Functional Overview

#### Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setup_qei(options, filter,maxcount)</code></td>
<td>Configures the QEI module.</td>
</tr>
<tr>
<td><code>qei_status()</code></td>
<td>Returns the status of the QEI module.</td>
</tr>
<tr>
<td><code>qei_set_count(value)</code></td>
<td>Write a 16-bit value to the position counter.</td>
</tr>
<tr>
<td><code>qei_get_count()</code></td>
<td>Reads the current 16-bit value of the position counter.</td>
</tr>
</tbody>
</table>

#### Relevant Preprocessor:

None

#### Relevant Interrupts:

`#INT_QEI` - Interrupt on rollover or underflow of the position counter

#### Relevant Include Files:

None, all functions built-in

#### Relevant `getenv()` parameters:

None

#### Example Code:

```c
int16 value;
setup_qei(QEI_MODE_X2| QEI_TIMER_INTERNAL, QEI_FILTER_DIV_2,QEI_FORWARD);  //Setup the QEI module

Value=qei_get_count();  //Read the count
```

### RS232 I/O

These functions and directives can be used for setting up and using RS232 I/O functionality.

#### Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getc()</code> or <code>getch()</code></td>
<td>Gets a character on the receive pin (from the specified stream in case of fgetc, stdin by default). Use KBHIT to check if the character is available.</td>
</tr>
<tr>
<td><code>getchar()</code> or <code>fgets()</code></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>gets()</code> or <code>fgets()</code></td>
<td>Gets a string on the receive pin (from the specified stream in case of <code>fgets</code>, STDIN by default). Use <code>getc</code> to receive each character until return is encountered.</td>
</tr>
<tr>
<td><code>putc()</code> or <code>putchar()</code> or <code>fputc()</code></td>
<td>Puts a character over the transmit pin (on the specified stream in the case of <code>fputc</code>, stdout by default)</td>
</tr>
<tr>
<td><code>puts()</code> or <code>fputs()</code></td>
<td>Puts a string over the transmit pin (on the specified stream in the case of <code>fputc</code>, stdout by default). Uses <code>putc</code> to send each character.</td>
</tr>
<tr>
<td><code>printf()</code> or <code>fprintf()</code></td>
<td>Prints the formatted string (on the specified stream in the case of <code>fprintf</code>, stdout by default). Refer to the <code>printf</code> help for details on format string.</td>
</tr>
<tr>
<td><code>kbhit()</code></td>
<td>Return true when a character is received in the buffer in case of hardware RS232 or when the first bit is sent on the RCV pin in case of software RS232. Useful for polling without waiting in <code>getc</code>.</td>
</tr>
<tr>
<td><code>setup_uart(baud,[stream])</code> or <code>setup_uart_speed(baud,[stream])</code></td>
<td>Used to change the baud rate of the hardware UART at run-time. Specifying stream is optional. Refer to the help for more advanced options.</td>
</tr>
<tr>
<td><code>assert(condition)</code></td>
<td>Checks the condition and if false prints the file name and line to STDERR. Will not generate code if <code>#DEFINE NODEBUG</code> is used.</td>
</tr>
<tr>
<td><code>perror(message)</code></td>
<td>Prints the message and the last system error to STDERR.</td>
</tr>
<tr>
<td><code>putc_send()</code> or <code>fputc_send()</code></td>
<td>When using transmit buffer, used to transmit data from buffer. See function description for more detail on when needed.</td>
</tr>
<tr>
<td><code>rcv_buffer_bytes()</code></td>
<td>When using receive buffer, returns the number of bytes in buffer that still need to be retrieved.</td>
</tr>
</tbody>
</table>
### Functional Overview

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tx_buffer_bytes()</strong></td>
<td>When using transmit buffer, returns the number of bytes in buffer that still need to be sent.</td>
</tr>
<tr>
<td><strong>tx_buffer_full()</strong></td>
<td>When using transmit buffer, returns TRUE if transmit buffer is full.</td>
</tr>
<tr>
<td><strong>receive_buffer_full()</strong></td>
<td>When using receive buffer, returns TRUE if receive buffer is full.</td>
</tr>
<tr>
<td><strong>tx_buffer_available()</strong></td>
<td>When using transmit buffer, returns number of characters that can be put into transmit buffer before it overflows.</td>
</tr>
</tbody>
</table>

**#useRS232**

Configures the compiler to support RS232 to specifications.

**Relevant Interrupts:**

- **INT_RDA**: Interrupt fires when the receive data available
- **INT_TBE**: Interrupt fires when the transmit data empty

Some chips have more than one hardware UART, and hence more interrupts.

**Relevant Include Files:**

None, all functions built-in

**Relevant getenv() parameters:**

- **UART**: Returns the number of UARTs on this PIC
- **AUART**: Returns true if this UART is an advanced UART
- **UART_RX**: Returns the receive pin for the first UART on this PIC (see PIN_XX)
- **UART_TX**: Returns the transmit pin for the first UART on this PIC
- **UART2_RX**: Returns the receive pin for the second UART on this PIC
- **UART2_TX**: Returns the pin TX – Returns the transmit pin for the second UART on this PIC
Example Code:

```c
/*configure and enable uart, use first hardware UART on PIC*/
#use rs232(uart1, baud=9600)

/* print a string*/
printf("enter a character");

/* get a character*/
if (kbhit()) //check if a character
has been received
c=getc(); //read character from UART
```

### RTCC

The Real Time Clock and Calendar (RTCC) module is intended for applications where accurate time must be maintained for extended periods of time with minimum or no intervention from the CPU. The key features of the module are:

- Time: Hour, Minute and Seconds.
- 24-hour format (Military Time)
- Calendar: Weekday, Date, Month and Year.
- Alarm Configurable.
- Requirements: External 32.768 kHz Clock Crystal.

#### Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setup_rtc (options, calibration);</code></td>
<td>This will setup the RTCC module for operation and also allows for calibration setup.</td>
</tr>
<tr>
<td><code>rtc_write(rtc_time_t datetime)</code></td>
<td>Writes the date and time to the RTCC module.</td>
</tr>
<tr>
<td><code>rtc_read(rtctime_t datetime)</code></td>
<td>Reads the current value of Time and Date from the RTCC module.</td>
</tr>
<tr>
<td><code>setup_rtc_alarm(options, mask, repeat);</code></td>
<td>Configures the alarm of the RTCC module.</td>
</tr>
<tr>
<td><code>rtc_alarm_write(rtctime_t datetime);</code></td>
<td>Writes the date and time to the alarm in the RTCC module.</td>
</tr>
<tr>
<td><code>rtc_alarm_read(rtctime_t)</code></td>
<td>Reads the date and time to the alarm in the RTCC</td>
</tr>
</tbody>
</table>
Functional Overview

### Relevant Preprocessor:
*None*

### Relevant Interrupts:
- **#INT_RTC** Interrupt on Alarm Event or half alarm frequency.

### Relevant Include Files:
*None, all functions built-in*

### Relevant getenv() parameters:
*None*

### Example Code:

```c
setup_rtc(RTC_ENABLE | RTC_OUTPUT_SECONDS, 0x00);  // Enable RTCC module with seconds clock and no calibration.
rtc_write(datetime);  // Write the value of Date and Time to the RTC module
rtc_read(datetime);   // Reads the value to a structure time_t.
```

### RTOS

These functions control the operation of the CCS Real Time Operating System (RTOS). This operating system is cooperatively multitasking and allows for tasks to be scheduled to run at specified time intervals. Because the RTOS does not use interrupts, the user must be careful to make use of the rtos_yield() function in every task so that no one task is allowed to run forever.

### Relevant Functions:

- **rtos_run()**
  - Begins the operation of the RTOS. All task management tasks are implemented by this function.

- **rtos_terminate()**
  - This function terminates the operation of the RTOS and returns operation to the original program. Works as a return from the rtos_run() function.

- **rtos_enable(task)**
  - Enables one of the RTOS tasks. Once a task is enabled, the rtos_run() function will call the task when its time occurs. The parameter to this function is the name of task to be enabled.

- **rtos_disable(task)**
  - Disables one of the RTOS tasks. Once a task is
disabled, the rtos_run() function will not call this task until it is enabled using rtos_enable(). The parameter to this function is the name of the task to be disabled.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtos_msg_poll()</td>
<td>Returns true if there is data in the task's message queue.</td>
</tr>
<tr>
<td>rtos_msg_read()</td>
<td>Returns the next byte of data contained in the task's message queue.</td>
</tr>
<tr>
<td>rtos_msg_send(task,byte)</td>
<td>Sends a byte of data to the specified task. The data is placed in the receiving task's message queue.</td>
</tr>
<tr>
<td>rtos_yield()</td>
<td>Called with in one of the RTOS tasks and returns control of the program to the rtos_run() function. All tasks should call this function when finished.</td>
</tr>
<tr>
<td>rtos_signal(sem)</td>
<td>Increments a semaphore which is used to broadcast the availability of a limited resource.</td>
</tr>
<tr>
<td>rtos_wait(sem)</td>
<td>Waits for the resource associated with the semaphore to become available and then decrements to semaphore to claim the resource.</td>
</tr>
<tr>
<td>rtosAwait(expre)</td>
<td>Will wait for the given expression to evaluate to true before allowing the task to continue.</td>
</tr>
<tr>
<td>rtos_overrun(task)</td>
<td>Will return true if the given task over ran its alloted time.</td>
</tr>
<tr>
<td>rtos_stats(task,stat)</td>
<td>Returns the specified statistic about the specified task. The statistics include the minimum and maximum times for the task to run and the total time the task has spent running.</td>
</tr>
</tbody>
</table>

Relevant Preprocessor:

#USE RTOS(options)  This directive is used to specify several different RTOS attributes including the timer to use, the minor cycle time and whether or not statistics should be enabled.

#TASK(options)       This directive tells the compiler that the following function is to be an RTOS task.

#TASK                specifies the rate at which the task should be called, the maximum time the task shall be allowed to run, and how large it's queue should be

Relevant Interrupts:
none
**Functional Overview**

Relevant Include Files:

```none
All functions are built in
```

Relevant getenv() Parameters:

```none
None
```

Example Code:

```c
#USE
RTOS(timer=0,minor_cycle=20ms)

... int sem;

... #TASK(rate=1s,max=20ms,queue=5)
void task_name();
rtos_run();
rtos.terminate();
rtos_enable(task_name);
rtos_disable(task_name);
rtos_msg_send(task_name,5);
rtos.yield();
rtos.signal(sem);
```

For more information on the CCS RTOS please

**SPI**

SPI™ is a fluid standard for 3 or 4 wire, full duplex communications named by Motorola. Most PIC devices support most common SPI™ modes. CCS provides a support library for taking advantage of both hardware and software based SPI™ functionality. For software support, see #USE SPI.

Relevant Functions:

```c
setup_spi(mode)
setup_spi2(mode)
```

Configure the hardware SPI to the specified mode. The mode configures setup_spi2(mode) thing such as master
**setup_spi3 (mode)**
**setup_spi4 (mode)**
or slave mode, clock speed and clock/data trigger configuration.

**Note:** for devices with dual SPI interfaces a second function, `setup_spi2()`, is provided to configure the second interface.

**spi_data_is_in()**
Returns TRUE if the SPI receive buffer has a byte of data.

**spi_data_is_in2()**

**spi_write(value)**
**spi_write2(value)**
Transmits the value over the SPI interface. This will cause the data to be clocked out on the SDO pin.

**spi_read(value)**
**spi_read2(value)**
Performs an SPI transaction, where the value is clocked out on the SDO pin and data clocked in on the SDI pin is returned. If you just want to clock in data then you can use `spi_read()` without a parameter.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
```
#int_ssp
#int_ssp2
Transaction (read or write) has completed on the indicated peripheral.
```
```
#int_spi1
#int_spi2
Interrupts on activity from the first SPI module
Interrupts on activity from the second SPI module
```

**Relevant Include Files:**
None, all functions built-in to the compiler.

**Relevant getenv() Parameters:**
SPI
Returns TRUE if the device has an SPI peripheral

**Example Code:**
```c
//configure the device to be a master, data transmitted on H-to-L clock transition
setup_spi(SPI_MASTER | SPI_H_TO_L | SPI_CLK_DIV_16);

spi_write(0x80);       //write 0x80 to SPI device
value=spi_read();      //read a value from the SPI device
value=spi_read(0x80);  //write 0x80 to SPI device the same time you are reading a value.
```
Timers

The 16-bit DSC and MCU families implement 16 bit timers. Many of these timers may be concatenated into a hybrid 32 bit timer. Also, one timer may be configured to use a low power 32.768 kHz oscillator which may be used as a real time clock source.

Timer1 is a 16 bit timer. It is the only timer that may not be concatenated into a hybrid 32 bit timer. However, it alone may use a synchronous external clock. This feature may be used with a low power 32.768 kHz oscillator to create a real-time clock source.

Timers 2 through 9 are 16 bit timers. They may use external clock sources only asynchronously and they may not act as low power real time clock sources. They may however be concatenated into 32 bit timers. This is done by configuring an even numbered timer (timer 2, 4, 6 or 8) as the least significant word, and the corresponding odd numbered timer (timer 3, 5, 7 or 9, respectively) as the most significant word of the new 32 bit timer.

Timer interrupts will occur when the timer overflows. Overflow will happen when the timer surpasses its period, which by default is 0xFFFF. The period value may be changed when using setup_timer_X.

Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup_timer_X()</td>
<td>Configures the timer peripheral. X may be any valid timer for the target device. Consult the target datasheet or use getenv to find the valid timers.</td>
</tr>
<tr>
<td>get_timerX()</td>
<td>Retrieves the current 16 bit value of the timer.</td>
</tr>
<tr>
<td>get_timerXY()</td>
<td>Gets the 32 bit value of the concatenated timers X and Y (where XY may only be 23, 45, 67, 89)</td>
</tr>
<tr>
<td>set_timerX()</td>
<td>Sets the value of timerX</td>
</tr>
<tr>
<td>set_timerXY()</td>
<td>Sets the 32 bit value of the concatenated timers X and Y (where XY may only be 23, 45, 67, 89)</td>
</tr>
</tbody>
</table>

Relevant Preprocessor:
None

Relevant Interrupts:

#define int_timerX Interrupts on timer overflow (period match). X is any valid timer number.

*When using a 32-bit timer, the odd numbered timer-interrupt of the hybrid timer must be used. (i.e. when using 32-bit Timer23, #int_timer3)

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMERX</td>
<td>Returns 1 if the device has the timer peripheral X. X may be 1 - 9</td>
</tr>
</tbody>
</table>

Example Code:

```c
/* Setup timer1 as an external real time clock that increments every 16 clock cycles */
setup_timer1(T1_EXTERNAL_RTC | T2_DIV_BY_16);

/* Setup timer2 as a timer that increments on every instruction cycle and has a period of 0x0100 */
setup_timer2(TMR_INTERNAL, 0x0100);
byte value = 0x00;
value = get_timer2();  // retrieve the current value of timer2
```

**TimerA**

These options lets the user configure and use timerA. It is available on devices with Timer A hardware. The clock/counter is 8 bit. It counts up and also provides interrupt on overflow. The options available are listed in the device's header file.

Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup_timer_A(mode)</td>
<td>Disable or sets the source and prescale for timerA</td>
</tr>
<tr>
<td>set_timerA(value)</td>
<td>Initializes the timerA clock/counter</td>
</tr>
<tr>
<td>value=get_timerA()</td>
<td>Returns the value of the timerA clock/counter</td>
</tr>
</tbody>
</table>

Relevant Preprocessor:

**None**

Relevant Interrupts:

- **INT_TIMERA**
  - Interrupt fires when timerA overflows

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMERA</td>
<td>Returns 1 if the device has timerA</td>
</tr>
</tbody>
</table>

Example Code:

```c
setup_timer_A(TA_OFF);  // disable timerA
or
```

94
setup_timer_A //sets the internal clock as source
(TA_INTERNAL | TA_DIV_8); //and prescale as 8. At 20MHz timerA will increment
//every 1.6us in this setup and overflows every
//409.6us
set_timerA(0); //this sets timerA register to 0
time=get_timerA(); //this will read the timerA register value

TimerB

These options lets the user configure and use timerB. It is available on devices with TimerB hardware. The clock/counter is 8 bit. It counts up and also provides interrupt on overflow. The options available are listed in the device’s header file.

Relevant Functions:

| setup_timer_B(mode) | Disable or sets the source and prescale for timerB |
| set_timerB(value)   | Initializes the timerB clock/counter |
| value=get_timerB()  | Returns the value of the timerB clock/counter |

Relevant Preprocessor: None

Relevant Interrupts : INT_TIMERB
Interrupt fires when timerB overflows

Relevant Include Files: None, all functions built-in

Relevant getenv() parameters:

| TIMERB | Returns 1 if the device has timerB |

Example Code:

setup_timer_B(TB_OFF); //disable timerB
or

setup_timer_B //sets the internal clock as source
(TB_INTERNAL | TB_DIV_8); //and prescale as 8. At 20MHz timerB will increment
//every 1.6us in this setup and overflows every
//409.6us
set_timerB(0); //this sets timerB register to 0
time=get_timerB(); //this will read the timerB register value
## Voltage Reference

These functions configure the voltage reference module. These are available only in the supported chips.

### Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup_vref(mode</td>
<td>value)</td>
</tr>
</tbody>
</table>

### Relevant Preprocessor:

None

### Relevant Interrupts:

None

### Relevant Include Files:

None, all functions built-in

### Relevant getenv() parameters:

VREF Returns 1 if the device has VREF

### Example code: for PIC12F675

```c
#INT_COMP  //comparator interrupt handler
void isr() {
    safe_conditions = FALSE;
    printf("WARNING!!!! Voltage level is above 3.6V. \r\n");
}

setup_comparator(A1_VR_OUT_ON_A2) //sets 2 comparators(A1 and VR and A2 as output) {
    setup_vref(VREF_HIGH | 15) //sets 3.6(vdd * value/32 + vdd/4) if vdd is 5.0V
    enable_interrunts(INT_COMP); // enable the comparator interrupt
    enable_interrunts(GLOBAL); //enable global interrupts
}
```
# WDT or Watch Dog Timer

Different chips provide different options to enable/disable or configure the WDT.

### Relevant Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setup_wdt()</code></td>
<td>Enables/disables the wdt or sets the prescalar.</td>
</tr>
<tr>
<td><code>restart_wdt()</code></td>
<td>Restarts the wdt, if wdt is enables this must be periodically called to prevent a timeout reset.</td>
</tr>
</tbody>
</table>

For PCB/PCM chips it is enabled/disabled using WDT or NOWDT fuses whereas on PCH device it is done using the `setup_wdt` function.

The timeout time for PCB/PCM chips are set using the `setup_wdt` function and on PCH using fuses like WDT16, WDT256 etc. RESTART_WDT when specified in `#USE DELAY`, `#USE I2C` and `#USE RS232` statements like this `#USE DELAY(clock=20000000, restart_wdt)` will cause the wdt to restart if it times out during the delay or `I2C_READ` or `GETC`.

### Relevant Preprocessor:

<table>
<thead>
<tr>
<th>Preprocessor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#FUSES WDT/NOWDT</code></td>
<td>Enabled/Disables wdt in PCB/PCM devices</td>
</tr>
<tr>
<td><code>#FUSES WDT16</code></td>
<td>Sets up the timeout time in PCH devices</td>
</tr>
</tbody>
</table>

### Relevant Interrupts: None

### Relevant Include Files: None

### Relevant `getenv()` parameters: None

### Example Code:

For PIC16F877

```c
#define wdt _setup_wdt(WDT_2304MS);
while(true){
    _restart_wdt();
    _perform_activity();
}
```

For PIC18F452

```c
#define WDT1
_setup_wdt(WDT_ON);
while(true){
    _restart_wdt();
    _perform_activity();
}
```

Some of the PCB chips are share the WDT prescalar bits with timer0 so the WDT prescalar constants can be used with `setup_counters` or `setup_timer0` or `setup_wdt` functions.
**interrupt_enabled()**

This function checks the interrupt enabled flag for the specified interrupt and returns TRUE if set.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>interrupt_enabled(interrupt);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>interrupt- constant specifying the interrupt</td>
</tr>
<tr>
<td>Returns</td>
<td>Boolean value</td>
</tr>
<tr>
<td>Function</td>
<td>The function checks the interrupt enable flag of the specified interrupt and returns TRUE when set.</td>
</tr>
<tr>
<td>Availability</td>
<td>Devices with interrupts</td>
</tr>
<tr>
<td>Requires</td>
<td>Interrupt constants defined in the device's .h file.</td>
</tr>
</tbody>
</table>
| Examples              | if(interrupt_enabled(INT_RDA))
                        |    disable_interrupt(INT_RDA); |
| Example Files         | None |
| Also see              | DISABLE_INTERRUPTS(), Interrupts Overview, CLEAR_INTERRUPT(), ENABLE_INTERRUPTS(), INTERRUPT_ACTIVE() |

**Stream I/O**

**Syntax:**

```
#include <ios.h> is required to use any of the ios identifiers.
```

**Output:**

```
output:
stream << variable_or_constant_or_manipulator ;
```

one or more repeats

`stream` may be the name specified in the #use RS232 stream= option
or for the default stream use cout.

`stream` may also be the name of a char array. In this case the data is written to the array with a 0 terminator.
**stream** may also be the name of a function that accepts a single char parameter. In this case the function is called for each character to be output.

**variables/constants**: May be any integer, char, float or fixed type. Char arrays are output as strings and all other types are output as an address of the variable.

**manipulators**:  
- `hex` - Hex format numbers  
- `dec` - Decimal format numbers (default)  
- `setprecision(x)` - Set number of places after the decimal point  
- `setw(x)` - Set total number of characters output for numbers  
- `boolalpha` - Output int1 as true and false  
- `noboolalpha` - Output int1 as 1 and 0 (default)  
- `fixed` - Floats in decimal format (default)  
- `scientific` - Use E notation  
- `iosdefault` - All manipulators to default settings  
- `endl` - Output CR/LF  
- `ends` - Outputs a null (`\000`)  

**Examples:**

```cpp
cout << "Value is " << hex << data << endl;
cout << "Price is $" << setw(4) << setprecision(2) << cost << endl;
lcdputc << \f << setw(3) << count << "min " << min << "max";
string1 << setprecision(1) << sum / count;
string2 << x << ',' << y;
```

**Input:**

```cpp
stream >> variable_or_constant_or_manipulator;
```

one or more repeats

*stream* may be the name specified in the `#use RS232 stream=` option  
or for the default stream use `cin`.  

*stream* may also be the name of a char array. In this case the data is read from the array up to the 0 terminator.  

*stream* may also be the name of a function that returns a
single char and has no parameters. In this case the function is called for each character to be input. Make sure the function returns a \r to terminate the input statement.

variables/constants: May be any integer, char, float or fixed type. Char arrays are input as strings. Floats may use the E format. Reading of each item terminates with any character not valid for the type. Usually items are separated by spaces. The termination character is discarded. At the end of any stream input statement characters are read until a return (\r) is read. No termination character is read for a single char input.

manipulators:
- hex - Hex format numbers
- dec - Decimal format numbers (default)
- noecho - Suppress echoing
- strspace - Allow spaces to be input into strings
- nostrspace - Spaces terminate string entry (default)
- iosdefault - All manipulators to default settings

Examples:
```cpp
cout << "Enter number: ";
cin >> value;
cout << "Enter title: ";
cin >> strspace >> title;
cin >> data[i].recordid >> data[i].xpos >> data[i].ypos >> data[i].sample ;
string1 >> data;
lcdputc << "\fEnter count";
lcdputc << keypadgetc >> count;  // read from keypad, echo to lcd
```

only works with functions.  // This syntax // user defined
Pre-processor directives all begin with a # and are followed by a specific command. Syntax is dependent on the command. Many commands do not allow other syntactical elements on the remainder of the line. A table of commands and a description is listed on the previous page.

Several of the pre-processor directives are extensions to standard C. C provides a pre-processor directive that compilers will accept and ignore or act upon the following data. This implementation will allow any pre-processor directives to begin with #PRAGMA. To be compatible with other compilers, this may be used before non-standard features.

Examples:
Both of the following are valid

```c
#INLINE
#PRAGMA INLINE
```

__address__

A predefined symbol __address__ may be used to indicate a type that must hold a program memory address.

For example:

```c
___address__ testa = 0x1000 //will allocate 16 bits for
test a and

//initialize to 0x1000
```
__attribute__x

**Syntax:**  __attribute__x

**Elements:**  x is the attribute you want to apply. Valid values for x are as follows:

((packed))
By default each element in a struct or union are padded to be evenly spaced by the size of 'int'. This is to prevent an address fault when accessing an element of struct. See the following example:

```c
struct
{
    int8 a;
    int16 b;
} test;
```

On architectures where 'int' is 16bit (such as dsPIC or PIC24 PICmicrocontrollers), 'test' would take 4 bytes even though it is comprised of 3 bytes. By applying the 'packed' attribute to this struct then it would take 3 bytes as originally intended:

```c
struct __attribute__((packed))
{
    int8 a;
    int16 b;
} test;
```

Care should be taken by the user when accessing individual elements of a packed struct – creating a pointer to 'b' in 'test' and attempting to dereference that pointer would cause an address fault. Any attempts to read/write 'b' should be done in context of 'test' so the compiler knows it is packed:

```c
test.b = 5;
```

((aligned(y))
By default the compiler will allocate a variable in the first free memory location. The aligned attribute will force the compiler to allocate a location for the specified variable at a location that is modulus of the y parameter. For example:

```c
int8 array[256] __attribute__((aligned(0x1000)));
```
This will tell the compiler to try to place 'array' at either 0x0, 0x1000, 0x2000, 0x3000, 0x4000, etc.

**Purpose** To alter some specifics as to how the compiler operates

**Examples:**

```c
struct __attribute__((packed))
{
```
```c
int8 a;
int8 b;
} test;
int8 array[256] __attribute__((aligned(0x1000)));
```

**Example Files:** None

### #asm  #endasm  #asm asis

**Syntax:** #ASM or #ASM ASIS code #ENDASM

**Elements:** code is a list of assembly language instructions

**Examples:**

```asm
int find_parity(int data){

  int count;
  #asm
  MOV #0x08, W0
  MOV W0, count
  CLR W0
  loop:
  XOR.B data,W0
  RRC data,W0
  DEC count,F
  BRA NZ, loop
  MOV #0x01,W0
  ADD count,F
  MOV count, W0
  MOV W0. _RETURN_
  #endasm
}
```

**Example Files:** FFT.c

**Also See:** None
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Destination</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa+Wb</td>
</tr>
<tr>
<td>ADD</td>
<td>f,W</td>
<td>W0 = f+Wd</td>
</tr>
<tr>
<td>ADD</td>
<td>lit10,Wd</td>
<td>Wd = lit10+Wd</td>
</tr>
<tr>
<td>ADD</td>
<td>Wa,lit5,Wd</td>
<td>Wd = lit5+Wa</td>
</tr>
<tr>
<td>ADD</td>
<td>f,F</td>
<td>f = f+Wd</td>
</tr>
<tr>
<td>ADD</td>
<td>acc</td>
<td>Acc = AccA+AccB</td>
</tr>
<tr>
<td>ADD</td>
<td>Wd,{lit4},acc</td>
<td>Acc = Acc+(Wa shifted slit4)</td>
</tr>
<tr>
<td>ADD.B</td>
<td>lit10,Wd</td>
<td>Wd = lit10+Wd (byte)</td>
</tr>
<tr>
<td>ADD.B</td>
<td>Wd,{lit4},acc</td>
<td>Acc = Acc+(Wa shifted slit4)</td>
</tr>
<tr>
<td>ADD.B</td>
<td>lit10,Wd</td>
<td>Wd = lit10+Wd (byte)</td>
</tr>
<tr>
<td>ADD.B</td>
<td>f,F</td>
<td>f = f+Wd (byte)</td>
</tr>
<tr>
<td>ADD.B</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa+Wb (byte)</td>
</tr>
<tr>
<td>ADD.B</td>
<td>Wa,lit5,Wd</td>
<td>Wd = lit5+Wa (byte)</td>
</tr>
<tr>
<td>ADD.B</td>
<td>f,W</td>
<td>W0 = f+Wd (byte)</td>
</tr>
<tr>
<td>ADDC</td>
<td>f,W</td>
<td>Wd = f+Wa+C</td>
</tr>
<tr>
<td>ADDC</td>
<td>lit10,Wd</td>
<td>Wd = lit10+Wd+C</td>
</tr>
<tr>
<td>ADDC</td>
<td>Wa,lit5,Wd</td>
<td>Wd = lit5+Wa+C</td>
</tr>
<tr>
<td>ADDC</td>
<td>f,F</td>
<td>Wd = f+Wa+C</td>
</tr>
<tr>
<td>ADDC</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa+Wb+C</td>
</tr>
<tr>
<td>ADDC.B</td>
<td>lit10,Wd</td>
<td>Wd = lit10+Wd+C (byte)</td>
</tr>
<tr>
<td>ADDC.B</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa+Wb+C (byte)</td>
</tr>
<tr>
<td>ADDC.B</td>
<td>Wa,lit5,Wd</td>
<td>Wd = lit5+Wa+C (byte)</td>
</tr>
<tr>
<td>ADDC.B</td>
<td>f,W</td>
<td>Wd = f+Wa+C (byte)</td>
</tr>
<tr>
<td>AND</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa.&amp;.Wb</td>
</tr>
<tr>
<td>AND</td>
<td>lit10,Wd</td>
<td>Wd = lit10.&amp;.Wd</td>
</tr>
<tr>
<td>AND</td>
<td>f,W</td>
<td>W0 = f.&amp;.Wa</td>
</tr>
<tr>
<td>AND</td>
<td>f,F</td>
<td>f = f.&amp;.Wa</td>
</tr>
<tr>
<td>AND</td>
<td>Wa,lit5,Wd</td>
<td>Wd = lit5.&amp;.Wa</td>
</tr>
<tr>
<td>AND.B</td>
<td>f,W</td>
<td>W0 = f.&amp;.Wa (byte)</td>
</tr>
<tr>
<td>AND.B</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa.&amp;.Wb (byte)</td>
</tr>
<tr>
<td>AND.B</td>
<td>lit10,Wd</td>
<td>Wd = lit10.&amp;.Wd (byte)</td>
</tr>
<tr>
<td>AND.B</td>
<td>f,F</td>
<td>f = f.&amp;.Wa (byte)</td>
</tr>
<tr>
<td>AND.B</td>
<td>Wa,lit5,Wd</td>
<td>Wd = lit5.&amp;.Wa (byte)</td>
</tr>
<tr>
<td>ASR</td>
<td>f,W</td>
<td>W0 = f &gt;&gt; 1 arithmetic</td>
</tr>
<tr>
<td>ASR</td>
<td>f,F</td>
<td>f = f &gt;&gt; 1 arithmetic</td>
</tr>
<tr>
<td>ASR</td>
<td>Wa,Wd</td>
<td>Wd = Wa &gt;&gt; 1 arithmetic</td>
</tr>
<tr>
<td>ASR</td>
<td>Wa,lit4,Wd</td>
<td>Wd = Wa &gt;&gt; lit4 arithmetic</td>
</tr>
<tr>
<td>ASR</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa &gt;&gt; Wb arithmetic</td>
</tr>
<tr>
<td>ASR.B</td>
<td>f,F</td>
<td>f = f &gt;&gt; 1 arithmetic (byte)</td>
</tr>
<tr>
<td>ASR.B</td>
<td>f,W</td>
<td>W0 = f &gt;&gt; 1 arithmetic (byte)</td>
</tr>
<tr>
<td>ASR.B</td>
<td>Wa,Wd</td>
<td>Wd = Wa &gt;&gt; 1 arithmetic (byte)</td>
</tr>
<tr>
<td>BCLR</td>
<td>f,B</td>
<td>f.bit = 0</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td><strong>BCLR</strong></td>
<td>Wd,B</td>
<td>Wa.bit = 0</td>
</tr>
<tr>
<td><strong>BCLR.B</strong></td>
<td>Wd,B</td>
<td>Wa.bit = 0 (byte)</td>
</tr>
<tr>
<td><strong>BRA</strong></td>
<td>a</td>
<td>Branch unconditionally</td>
</tr>
<tr>
<td><strong>BRA BZ</strong></td>
<td>a</td>
<td>Branch if Zero</td>
</tr>
<tr>
<td><strong>BRA C</strong></td>
<td>a</td>
<td>Branch if Carry (no borrow)</td>
</tr>
<tr>
<td><strong>BRA GE</strong></td>
<td>a</td>
<td>Branch if greater than or equal</td>
</tr>
<tr>
<td><strong>BRA GEU</strong></td>
<td>a</td>
<td>Branch if unsigned greater than or equal</td>
</tr>
<tr>
<td><strong>BRA GT</strong></td>
<td>a</td>
<td>Branch if greater than</td>
</tr>
<tr>
<td><strong>BRA GTU</strong></td>
<td>a</td>
<td>Branch if unsigned greater than</td>
</tr>
<tr>
<td><strong>BRA LE</strong></td>
<td>a</td>
<td>Branch if less than or equal</td>
</tr>
<tr>
<td><strong>BRA LEU</strong></td>
<td>a</td>
<td>Branch if unsigned less than or equal</td>
</tr>
<tr>
<td><strong>BRA LT</strong></td>
<td>a</td>
<td>Branch if less than</td>
</tr>
<tr>
<td><strong>BRA LTU</strong></td>
<td>a</td>
<td>Branch if unsigned less than</td>
</tr>
<tr>
<td><strong>BRA N</strong></td>
<td>a</td>
<td>Branch if negative</td>
</tr>
<tr>
<td><strong>BRA NC</strong></td>
<td>a</td>
<td>Branch if not carry (Borrow)</td>
</tr>
<tr>
<td><strong>BRA NN</strong></td>
<td>a</td>
<td>Branch if not negative</td>
</tr>
<tr>
<td><strong>BRA NOV</strong></td>
<td>a</td>
<td>Branch if not Overflow</td>
</tr>
<tr>
<td><strong>BRA NZ</strong></td>
<td>a</td>
<td>Branch if not Zero</td>
</tr>
<tr>
<td><strong>BRA OA</strong></td>
<td>a</td>
<td>Branch if Accumulator A overflow</td>
</tr>
<tr>
<td><strong>BRA OB</strong></td>
<td>a</td>
<td>Branch if Accumulator B overflow</td>
</tr>
<tr>
<td><strong>BRA OV</strong></td>
<td>a</td>
<td>Branch if Overflow</td>
</tr>
<tr>
<td><strong>BRA SA</strong></td>
<td>a</td>
<td>Branch if Accumulator A Saturate</td>
</tr>
<tr>
<td><strong>BRA SB</strong></td>
<td>a</td>
<td>Branch if Accumulator B Saturate</td>
</tr>
<tr>
<td><strong>BRA Z</strong></td>
<td>a</td>
<td>Branch if Zero</td>
</tr>
<tr>
<td><strong>BREAK</strong></td>
<td></td>
<td>ICD Break</td>
</tr>
<tr>
<td><strong>BSET</strong></td>
<td>Wd,B</td>
<td>Wa.bit = 1</td>
</tr>
<tr>
<td><strong>BSET</strong></td>
<td>f,B</td>
<td>f.bit = 1</td>
</tr>
<tr>
<td><strong>BSET.B</strong></td>
<td>Wd,B</td>
<td>Wa.bit = 1 (byte)</td>
</tr>
<tr>
<td><strong>BSW.C</strong></td>
<td>Wa,Wd</td>
<td>Wa.Wb = C</td>
</tr>
<tr>
<td><strong>BSW.Z</strong></td>
<td>Wa,Wd</td>
<td>Wa.Wb = Z</td>
</tr>
<tr>
<td><strong>BTG</strong></td>
<td>Wd,B</td>
<td>Wa.bit = ~Wa.bit</td>
</tr>
<tr>
<td><strong>BTG</strong></td>
<td>f,B</td>
<td>f.bit = ~f.bit</td>
</tr>
<tr>
<td><strong>BTG.B</strong></td>
<td>Wd,B</td>
<td>Wa.bit = ~Wa.bit (byte)</td>
</tr>
<tr>
<td><strong>BTSC</strong></td>
<td>f,B</td>
<td>Skip if f.bit = 0</td>
</tr>
<tr>
<td><strong>BTSC</strong></td>
<td>Wd,B</td>
<td>Skip if Wa.bit4 = 0</td>
</tr>
<tr>
<td><strong>BTSS</strong></td>
<td>f,B</td>
<td>Skip if f.bit = 1</td>
</tr>
<tr>
<td><strong>BTSS</strong></td>
<td>Wd,B</td>
<td>Skip if Wa.bit = 1</td>
</tr>
<tr>
<td><strong>BTST</strong></td>
<td>f,B</td>
<td>Z = f.bit</td>
</tr>
<tr>
<td><strong>BTST.C</strong></td>
<td>Wa,Wd</td>
<td>C = Wa.Wb</td>
</tr>
<tr>
<td><strong>BTST.C</strong></td>
<td>Wd,B</td>
<td>C = Wa.bit</td>
</tr>
<tr>
<td><strong>BTST.Z</strong></td>
<td>Wd,B</td>
<td>Z = Wa.bit</td>
</tr>
<tr>
<td><strong>BTST.Z</strong></td>
<td>Wa,Wd</td>
<td>Z = Wa.Wb</td>
</tr>
<tr>
<td><strong>BTSTS</strong></td>
<td>f,B</td>
<td>Z = f.bit; f.bit = 1</td>
</tr>
<tr>
<td><strong>BTSTS.C</strong></td>
<td>Wd,B</td>
<td>C = Wa.bit; Wa.bit = 1</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>BTSTS.Z</strong></td>
<td>$Wd, B$</td>
<td></td>
</tr>
<tr>
<td><strong>CALL</strong></td>
<td>$a$ Call subroutine</td>
<td></td>
</tr>
<tr>
<td><strong>CALL</strong></td>
<td>$Wd$ Call $[Wa]$</td>
<td></td>
</tr>
<tr>
<td><strong>CLR</strong></td>
<td>$f,F$ $f = 0$</td>
<td></td>
</tr>
<tr>
<td><strong>CLR</strong></td>
<td>$acc, da, dc, pi$ $Acc = 0$; $prefetch = 0$</td>
<td></td>
</tr>
<tr>
<td><strong>CLR</strong></td>
<td>$f,W$ $W0 = 0$</td>
<td></td>
</tr>
<tr>
<td><strong>CLR</strong></td>
<td>$Wd$ $Wd = 0$</td>
<td></td>
</tr>
<tr>
<td><strong>CLR.B</strong></td>
<td>$f, W$ $W0 = 0$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CLR.B</strong></td>
<td>$Wd$ $Wd = 0$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CLR.B</strong></td>
<td>$f, F$ $f = 0$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CLRWDT</strong></td>
<td>Clear WDT</td>
<td></td>
</tr>
<tr>
<td><strong>COM</strong></td>
<td>$f, F$</td>
<td></td>
</tr>
<tr>
<td><strong>COM</strong></td>
<td>$f, W$</td>
<td></td>
</tr>
<tr>
<td><strong>COM</strong></td>
<td>$Wa, Wd$ $Wd = \sim Wa$</td>
<td></td>
</tr>
<tr>
<td><strong>COM.B</strong></td>
<td>$f, W$</td>
<td></td>
</tr>
<tr>
<td><strong>COM.B</strong></td>
<td>$Wa, Wd$ $Wd = \sim Wa$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>COM.B</strong></td>
<td>$f, F$</td>
<td></td>
</tr>
<tr>
<td><strong>CP</strong></td>
<td>$W, f$ Status set for $f - W0$</td>
<td></td>
</tr>
<tr>
<td><strong>CP</strong></td>
<td>$Wa, Wd$ Status set for $Wb \sim Wa$</td>
<td></td>
</tr>
<tr>
<td><strong>CP</strong></td>
<td>$Wd, lit5$ Status set for $Wa \sim lit5$</td>
<td></td>
</tr>
<tr>
<td><strong>CP.B</strong></td>
<td>$W, f$ Status set for $f - W0$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CP.B</strong></td>
<td>$Wa, Wd$ Status set for $Wb \sim Wa$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CP.B</strong></td>
<td>$Wd, lit5$ Status set for $Wa \sim lit5$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CP0</strong></td>
<td>$Wd$ Status set for $Wa \sim 0$</td>
<td></td>
</tr>
<tr>
<td><strong>CP0</strong></td>
<td>$W, f$ Status set for $f \sim 0$</td>
<td></td>
</tr>
<tr>
<td><strong>CP0.B</strong></td>
<td>$Wd$ Status set for $Wa \sim 0$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CP0.B</strong></td>
<td>$W, f$ Status set for $f \sim 0$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CPB</strong></td>
<td>$Wa, Wd$ Status set for $Wb \sim Wa \sim C$</td>
<td></td>
</tr>
<tr>
<td><strong>CPB</strong></td>
<td>$Wd, lit5$ Status set for $Wa \sim lit5 \sim C$</td>
<td></td>
</tr>
<tr>
<td><strong>CPB</strong></td>
<td>$Wa, Wd$ Status set for $Wb \sim Wa \sim C$</td>
<td></td>
</tr>
<tr>
<td><strong>CPB</strong></td>
<td>$W, f$ Status set for $f \sim W0 - C$</td>
<td></td>
</tr>
<tr>
<td><strong>CPB.B</strong></td>
<td>$Wa, Wd$ Status set for $Wb \sim Wa \sim C$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CPB.B</strong></td>
<td>$Wd, lit5$ Status set for $Wa \sim lit5 \sim C$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CPB.B</strong></td>
<td>$W, f$ Status set for $f \sim W0 - C$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CPSEQ</strong></td>
<td>$Wa, Wd$ Skip if $Wa = Wb$</td>
<td></td>
</tr>
<tr>
<td><strong>CPSEQ.B</strong></td>
<td>$Wa, Wd$ Skip if $Wa = Wb$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CPSGT</strong></td>
<td>$Wa, Wd$ Skip if $Wa &gt; Wb$</td>
<td></td>
</tr>
<tr>
<td><strong>CPSGT.B</strong></td>
<td>$Wa, Wd$ Skip if $Wa &gt; Wb$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CPSLT</strong></td>
<td>$Wa, Wd$ Skip if $Wa &lt; Wb$</td>
<td></td>
</tr>
<tr>
<td><strong>CPSLT.B</strong></td>
<td>$Wa, Wd$ Skip if $Wa &lt; Wb$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>CPSNE</strong></td>
<td>$Wa, Wd$ Skip if $Wa \neq Wb$</td>
<td></td>
</tr>
<tr>
<td><strong>CPSNE.B</strong></td>
<td>$Wa, Wd$ Skip if $Wa \neq Wb$ (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>DAW.B</strong></td>
<td>$Wd$ $Wa = \text{decimal adjust } Wa$</td>
<td></td>
</tr>
<tr>
<td><strong>DEC</strong></td>
<td>$Wa, Wd$ $Wd = Wa \sim 1$</td>
<td></td>
</tr>
<tr>
<td><strong>DEC</strong></td>
<td>$f, W$</td>
<td></td>
</tr>
<tr>
<td><strong>DEC</strong></td>
<td>$f, F$</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>DEC.B f, F</td>
<td>( f = f - 1 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>DEC.B f, W</td>
<td>( W0 = f - 1 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>DEC.B Wa, Wd</td>
<td>( Wd = Wa - 1 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>DEC2 Wa, Wd</td>
<td>( Wd = Wa - 2 )</td>
<td></td>
</tr>
<tr>
<td>DEC2 f, W</td>
<td>( W0 = f - 2 )</td>
<td></td>
</tr>
<tr>
<td>DEC2 f, F</td>
<td>( f = f - 2 )</td>
<td></td>
</tr>
<tr>
<td>DEC2.B Wa, Wd</td>
<td>( Wd = Wa - 2 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>DEC2.B f, W</td>
<td>( W0 = f - 2 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>DEC2.B f, F</td>
<td>( f = f - 2 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>DISI lit14</td>
<td>Disable Interrupts lit14 cycles</td>
<td></td>
</tr>
<tr>
<td>DIV.S Wa, Wd</td>
<td>Signed 16/16-bit integer divide</td>
<td></td>
</tr>
<tr>
<td>DIV.SD Wa, Wd</td>
<td>Signed 16/16-bit integer divide (dword)</td>
<td></td>
</tr>
<tr>
<td>DIV.U Wa, Wd</td>
<td>Unsigned 16/16-bit integer divide</td>
<td></td>
</tr>
<tr>
<td>DIV.UD Wa, Wd</td>
<td>Unsigned 16/16-bit integer divide (dword)</td>
<td></td>
</tr>
<tr>
<td>DIVF Wa, Wd</td>
<td>Signed 16/16-bit fractional divide</td>
<td></td>
</tr>
<tr>
<td>DO lit14, a</td>
<td>Do block lit14 times</td>
<td></td>
</tr>
<tr>
<td>DO Wd, a</td>
<td>Do block Wa times</td>
<td></td>
</tr>
<tr>
<td>ED Wd*Wd, acc, da, db</td>
<td>Euclidean Distance (No Accumulate)</td>
<td></td>
</tr>
<tr>
<td>EDAC Wd*Wd, acc, da, db</td>
<td>Euclidean Distance</td>
<td></td>
</tr>
<tr>
<td>EXCH Wa, Wd</td>
<td>Swap Wa and Wb</td>
<td></td>
</tr>
<tr>
<td>FBCL Wa, Wd</td>
<td>Find bit change from left (Msb) side</td>
<td></td>
</tr>
<tr>
<td>FEX</td>
<td>ICD Execute</td>
<td></td>
</tr>
<tr>
<td>FF1L Wa, Wd</td>
<td>Find first one from left (Msb) side</td>
<td></td>
</tr>
<tr>
<td>FF1R Wa, Wd</td>
<td>Find first one from right (Lsb) side</td>
<td></td>
</tr>
<tr>
<td>GOTO a</td>
<td>GoTo</td>
<td></td>
</tr>
<tr>
<td>GOTO Wd</td>
<td>GoTo [Wa]</td>
<td></td>
</tr>
<tr>
<td>INC f, W</td>
<td>( W0 = f + 1 )</td>
<td></td>
</tr>
<tr>
<td>INC Wa, Wd</td>
<td>( Wd = Wa + 1 )</td>
<td></td>
</tr>
<tr>
<td>INC f, F</td>
<td>( f = f + 1 )</td>
<td></td>
</tr>
<tr>
<td>INC.B Wa, Wd</td>
<td>( Wd = Wa + 1 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>INC.B f, F</td>
<td>( f = f + 1 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>INC.B f, W</td>
<td>( W0 = f + 1 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>INC2 f, W</td>
<td>( W0 = f + 2 )</td>
<td></td>
</tr>
<tr>
<td>INC2 Wa, Wd</td>
<td>( Wd = Wa + 2 )</td>
<td></td>
</tr>
<tr>
<td>INC2 f, F</td>
<td>( f = f + 2 )</td>
<td></td>
</tr>
<tr>
<td>INC2.B f, W</td>
<td>( W0 = f + 2 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>INC2.B f, F</td>
<td>( f = f + 2 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>INC2.B Wa, Wd</td>
<td>( Wd = Wa + 2 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>IOR lit10, Wd</td>
<td>( Wd = \text{lit10} )</td>
<td></td>
</tr>
<tr>
<td>IOR f, F</td>
<td>( f = f )</td>
<td></td>
</tr>
<tr>
<td>IOR f, W</td>
<td>( W0 = f )</td>
<td></td>
</tr>
<tr>
<td>IOR Wa, lit5, Wd</td>
<td>( Wd = Wa )</td>
<td></td>
</tr>
<tr>
<td>IOR Wa, Wb, Wd</td>
<td>( Wd = Wa )</td>
<td></td>
</tr>
<tr>
<td>IOR.B Wa, Wb, Wd</td>
<td>( Wd = Wa ) (byte)</td>
<td></td>
</tr>
<tr>
<td>IOR.B f, W</td>
<td>( W0 = f ) (byte)</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>IOR.B</td>
<td>( W_d = \text{lit10} \mid W_d ) (byte)</td>
<td></td>
</tr>
<tr>
<td>IOR.B</td>
<td>( W_d = W_a \cdot \text{lit5} ) (byte)</td>
<td></td>
</tr>
<tr>
<td>IOR.B</td>
<td>( f = f \mid W_a ) (byte)</td>
<td></td>
</tr>
<tr>
<td>LAC</td>
<td>( \text{Acc} = W_a \text{ shifted slit4} )</td>
<td></td>
</tr>
<tr>
<td>LNK</td>
<td>Allocate Stack Frame</td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>( W_0 = f \gg 1 )</td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>( W_d = W_a \gg \text{lit4} )</td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>( W_d = W_a \gg 1 )</td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>( f = f \gg 1 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>LSR.B</td>
<td>( W_d = W_b \gg W_a )</td>
<td></td>
</tr>
<tr>
<td>LSR.B</td>
<td>( W_0 = f \gg 1 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>LSR.B</td>
<td>( W_d = W_a \gg 1 ) (byte)</td>
<td></td>
</tr>
<tr>
<td>MAC</td>
<td>( \text{Acc} = \text{Acc} + W_a \cdot W_a ) {prefetch}</td>
<td></td>
</tr>
<tr>
<td>MAC</td>
<td>( \text{Acc} = \text{Acc} + W_a \cdot W_b; {W13} = \text{Acc}; {prefetch}</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( f = Wa )</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_0 = f )</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( f = f )</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( F = Wa )</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_d = [W_a + \text{Slit10}] )</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_d = W_f )</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_0 = f ) (byte)</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( f = f ) (byte)</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_d = \text{lit16} )</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_d = \text{lit8} ) (byte)</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( f = Wa ) (byte)</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_0 = f ) (byte)</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( f = f ) (byte)</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_d = [W_a + \text{Slit10}] ) (byte)</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_d = W_a ) (byte)</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_d = W_a \cdot W_a + 1 )</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>( W_d = W_a \cdot W_a + 1 )</td>
<td></td>
</tr>
<tr>
<td>MOV.SAC</td>
<td>Move ? to ? and ? to ?</td>
<td></td>
</tr>
<tr>
<td>MPY</td>
<td>( W_a \cdot W_c = W_a \cdot W_b )</td>
<td></td>
</tr>
<tr>
<td>MPY</td>
<td>( W_a \cdot W_c = \text{Square to Acc} )</td>
<td></td>
</tr>
<tr>
<td>MPY</td>
<td>( \text{Acc} = -W_a \cdot W_b )</td>
<td></td>
</tr>
<tr>
<td>MSC</td>
<td>( \text{Acc} = \text{Acc â€¢} \cdot W_a \cdot W_b )</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>( W_3 : W_2 = f \cdot W_a )</td>
<td></td>
</tr>
<tr>
<td>MUL.B</td>
<td>( W_3 : W_2 = f \cdot W_a ) (byte)</td>
<td></td>
</tr>
<tr>
<td>MUL.SS</td>
<td>( {W_d + 1,W_d} = \text{sign}(W_a) \cdot \text{sign}(W_b) )</td>
<td></td>
</tr>
<tr>
<td>MUL.SU</td>
<td>( {W_d + 1,W_d} = \text{sign}(W_a) \cdot \text{unsig}(W_b) )</td>
<td></td>
</tr>
<tr>
<td>MUL.SU</td>
<td>( {W_d + 1,W_d} = \text{sign}(W_a) \cdot \text{unsig}(\text{lit5}) )</td>
<td></td>
</tr>
<tr>
<td>MUL.US</td>
<td>( {W_d + 1,W_d} = \text{unsig}(W_a) \cdot \text{sign}(W_b) )</td>
<td></td>
</tr>
<tr>
<td>MUL.UU</td>
<td>( {W_d + 1,W_d} = \text{unsig}(W_a) \cdot \text{unsig}(W_b) )</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>MUL.UU</strong></td>
<td>( {W_d+1,W_d} = \text{unsigned}(W_a) * \text{unsigned}(\text{lit5}) )</td>
<td></td>
</tr>
<tr>
<td><strong>NEG</strong></td>
<td>( f = -f )</td>
<td></td>
</tr>
<tr>
<td><strong>PUSH</strong></td>
<td>Push ( W_a ) to TOS</td>
<td></td>
</tr>
<tr>
<td><strong>PUSH.D</strong></td>
<td>Push double ( W_a ) to ( W_a + 1 ) to TOS</td>
<td></td>
</tr>
<tr>
<td><strong>PUSH.S</strong></td>
<td>Push shadow registers</td>
<td></td>
</tr>
<tr>
<td><strong>PWRSAV</strong></td>
<td>Enter Power-saving mode ( \text{lit1} )</td>
<td></td>
</tr>
<tr>
<td><strong>RCALL</strong></td>
<td>Call (relative)</td>
<td></td>
</tr>
<tr>
<td><strong>RCALL</strong></td>
<td>Call ( W_a )</td>
<td></td>
</tr>
<tr>
<td><strong>REPEAT</strong></td>
<td>Repeat next instruction (( \text{lit14} + 1 )) times</td>
<td></td>
</tr>
<tr>
<td><strong>REPEAT</strong></td>
<td>Repeat next instruction (( W_a + 1 )) times</td>
<td></td>
</tr>
<tr>
<td><strong>RESET</strong></td>
<td>Reset</td>
<td></td>
</tr>
<tr>
<td><strong>RETFIE</strong></td>
<td>Return from interrupt enable</td>
<td></td>
</tr>
<tr>
<td><strong>RETLW</strong></td>
<td>Return; ( W_a = \text{lit10} )</td>
<td></td>
</tr>
<tr>
<td><strong>RETLW.B</strong></td>
<td>Return; ( W_a = \text{lit10} ) (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>RETURN</strong></td>
<td>Return</td>
<td></td>
</tr>
<tr>
<td><strong>RLC</strong></td>
<td>( W_d = \text{rotate left through Carry} W_a )</td>
<td></td>
</tr>
<tr>
<td><strong>RLC</strong></td>
<td>( f = \text{rotate left through Carry} f )</td>
<td></td>
</tr>
<tr>
<td><strong>RLC.B</strong></td>
<td>( f = \text{rotate left through Carry} f ) (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>RLNC</strong></td>
<td>( f = \text{rotate left} ) (no Carry) ( f )</td>
<td></td>
</tr>
<tr>
<td><strong>RLNC</strong></td>
<td>( W_0 = \text{rotate left} ) (no Carry) ( f )</td>
<td></td>
</tr>
<tr>
<td><strong>RLNC.B</strong></td>
<td>( f = \text{rotate left} ) (no Carry) ( f ) (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>RRC</strong></td>
<td>( f = \text{rotate right through Carry} f )</td>
<td></td>
</tr>
<tr>
<td><strong>RRC</strong></td>
<td>( W_0 = \text{rotate right through Carry} f )</td>
<td></td>
</tr>
<tr>
<td><strong>RRC.B</strong></td>
<td>( f = \text{rotate right through Carry} f ) (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>RRC.B</strong></td>
<td>( W_0 = \text{rotate right through Carry} f ) (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>RRNC</strong></td>
<td>( f = \text{rotate right} ) (no Carry) ( f )</td>
<td></td>
</tr>
<tr>
<td><strong>RRNC</strong></td>
<td>( W_0 = \text{rotate right} ) (no Carry) ( f )</td>
<td></td>
</tr>
<tr>
<td><strong>RRNC</strong></td>
<td>( W_0 = \text{rotate right} ) (no Carry) ( W_a )</td>
<td></td>
</tr>
<tr>
<td><strong>RRNC.B</strong></td>
<td>( f = \text{rotate right} ) (no Carry) ( f ) (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>RRNC.B</strong></td>
<td>( W_0 = \text{rotate right} ) (no Carry) ( W_a ) (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>RRNC.B</strong></td>
<td>( W_0 = \text{rotate right} ) (no Carry) ( W_a ) (byte)</td>
<td></td>
</tr>
<tr>
<td><strong>SAC</strong></td>
<td>( W_d = \text{Acc slt 4} )</td>
<td></td>
</tr>
<tr>
<td><strong>SAC.R</strong></td>
<td>( W_d = \text{Acc slt 4 with rounding} )</td>
<td></td>
</tr>
<tr>
<td><strong>SE</strong></td>
<td>( W_d = \text{sign-extended} W_a )</td>
<td></td>
</tr>
<tr>
<td><strong>SE</strong></td>
<td>( W_d = \text{sign-extended} W_a )</td>
<td></td>
</tr>
<tr>
<td><strong>SETM</strong></td>
<td>( W_d = 0xFFFF )</td>
<td></td>
</tr>
<tr>
<td><strong>SETM</strong></td>
<td>( W_0 = 0xFFFF )</td>
<td></td>
</tr>
<tr>
<td><strong>SETM.B</strong></td>
<td>( W_d = 0xFFFF ) (byte)</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>SETM.B</td>
<td>f,W</td>
<td>W0 = 0xFFFF (byte)</td>
</tr>
<tr>
<td>SETM.B</td>
<td>f,F</td>
<td>W0 = 0xFFFF (byte)</td>
</tr>
<tr>
<td>SFTAC</td>
<td>acc,Wd</td>
<td>Arithmetic shift Acc by (Wa)</td>
</tr>
<tr>
<td>SFTAC</td>
<td>acc,lit5</td>
<td>Arithmetic shift Acc by Slit6</td>
</tr>
<tr>
<td>SL</td>
<td>f,W</td>
<td>W0 = f &lt;&lt; 1</td>
</tr>
<tr>
<td>SL</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa &lt;&lt; Wb</td>
</tr>
<tr>
<td>SL</td>
<td>Wa,lit4,Wd</td>
<td>Wd = Wa &lt;&lt; lit4</td>
</tr>
<tr>
<td>SL</td>
<td>Wa,Wd</td>
<td>Wd = Wa &lt;&lt; 1</td>
</tr>
<tr>
<td>SL</td>
<td>f,F</td>
<td>f = f &lt;&lt; 1</td>
</tr>
<tr>
<td>SL.B</td>
<td>f,W</td>
<td>W0 = f &lt;&lt; 1 (byte)</td>
</tr>
<tr>
<td>SL.B</td>
<td>Wa,Wd</td>
<td>Wd = Wa &lt;&lt; 1 (byte)</td>
</tr>
<tr>
<td>SSTEP</td>
<td></td>
<td>ICD Single Step</td>
</tr>
<tr>
<td>SUB</td>
<td>f,F</td>
<td>f = f â€“ W0</td>
</tr>
<tr>
<td>SUB</td>
<td>f,W</td>
<td>W0 = f â€“ W0</td>
</tr>
<tr>
<td>SUB</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa â€“ Wb</td>
</tr>
<tr>
<td>SUB</td>
<td>Wa,lit5,Wd</td>
<td>Wd = Wa â€“ lit5</td>
</tr>
<tr>
<td>SUB</td>
<td>acc</td>
<td>Acc = AccA â€“ AccB</td>
</tr>
<tr>
<td>SUB</td>
<td>lit10,Wd</td>
<td>Wd = Wd â€“ lit10</td>
</tr>
<tr>
<td>SUB.B</td>
<td>Wa,lit5,Wd</td>
<td>Wd = Wa â€“ lit5 (byte)</td>
</tr>
<tr>
<td>SUB.B</td>
<td>lit10,Wd</td>
<td>Wd = Wd â€“ lit10 (byte)</td>
</tr>
<tr>
<td>SUB.B</td>
<td>f,W</td>
<td>W0 = f â€“ W0  (byte)</td>
</tr>
<tr>
<td>SUB.B</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa â€“ Wb (byte)</td>
</tr>
<tr>
<td>SUB.B</td>
<td>f,F</td>
<td>f = f â€“ W0   (byte)</td>
</tr>
<tr>
<td>SUBB</td>
<td>f,W</td>
<td>W0 = f â€“ W0 â€“ C</td>
</tr>
<tr>
<td>SUBB</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa â€“ Wb â€“ C</td>
</tr>
<tr>
<td>SUBB</td>
<td>f,F</td>
<td>f = f â€“ W0 â€“ C</td>
</tr>
<tr>
<td>SUBB</td>
<td>Wa,lit5,Wd</td>
<td>Wd = Wa â€“ lit5 - C</td>
</tr>
<tr>
<td>SUBB</td>
<td>lit10,Wd</td>
<td>Wd = Wd â€“ lit10 â€“ C</td>
</tr>
<tr>
<td>SUBB.B</td>
<td>lit10,Wd</td>
<td>Wd = Wd â€“ lit10 â€“ C (byte)</td>
</tr>
<tr>
<td>SUBB.B</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa â€“ Wb â€“ C (byte)</td>
</tr>
<tr>
<td>SUBB.B</td>
<td>f,F</td>
<td>f = f â€“ W0 â€“ C (byte)</td>
</tr>
<tr>
<td>SUBBR</td>
<td>Wa,lit5,Wd</td>
<td>Wd = lit5 â€“ Wa - C</td>
</tr>
<tr>
<td>SUBBR</td>
<td>f,W</td>
<td>W0 = W0 â€“ f â€“ C</td>
</tr>
<tr>
<td>SUBBR</td>
<td>f,F</td>
<td>f = W0 â€“ f â€“ C</td>
</tr>
<tr>
<td>SUBBR</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa â€“ Wb - C</td>
</tr>
<tr>
<td>SUBBR.B</td>
<td>f,F</td>
<td>f = W0 â€“ f â€“ C (byte)</td>
</tr>
<tr>
<td>SUBBR.B</td>
<td>f,W</td>
<td>W0 = W0 â€“ f â€“ C (byte)</td>
</tr>
<tr>
<td>SUBBR.B</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa â€“ Wb - C (byte)</td>
</tr>
<tr>
<td>SUBBR.B</td>
<td>Wa,lit5,Wd</td>
<td>Wd = lit5 â€“ Wa - C (byte)</td>
</tr>
<tr>
<td>SUBR</td>
<td>Wa,lit5,Wd</td>
<td>Wd = lit5 â€“ Wb</td>
</tr>
<tr>
<td>SUBR</td>
<td>f,F</td>
<td>f = W0 â€“ f</td>
</tr>
<tr>
<td>SUBR</td>
<td>Wa,Wb,Wd</td>
<td>Wd = Wa â€“ Wb</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>SUBR</td>
<td>$f, W$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_0 = W_0 \oplus f$</td>
<td></td>
</tr>
<tr>
<td>SUBR.B</td>
<td>$W_a, W_b, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = W_a \oplus W_b$ (byte)</td>
<td></td>
</tr>
<tr>
<td>SUBR.B</td>
<td>$f, F$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$f = W_0 \oplus f$ (byte)</td>
<td></td>
</tr>
<tr>
<td>SUBR.B</td>
<td>$W_a, \text{lit}5, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = \text{lit}5 \oplus W_b$ (byte)</td>
<td></td>
</tr>
<tr>
<td>SUBR.B</td>
<td>$f, W$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_0 = W_0 \oplus f$ (byte)</td>
<td></td>
</tr>
<tr>
<td>SWAP</td>
<td>$W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_a = \text{byte or nibble swap } W_a$</td>
<td></td>
</tr>
<tr>
<td>SWAP.B</td>
<td>$W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_a = \text{byte or nibble swap } W_a$ (byte)</td>
<td></td>
</tr>
<tr>
<td>TBLRDL</td>
<td>$W_a, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = \text{ROM}[W_a]$ for even ROM (byte)</td>
<td></td>
</tr>
<tr>
<td>TBLRDL.B</td>
<td>$W_a, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = \text{ROM}[W_a]$ for odd ROM (byte)</td>
<td></td>
</tr>
<tr>
<td>TBLWTH</td>
<td>$W_a, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{ROM}[W_a] = W_d$ for odd ROM (byte)</td>
<td></td>
</tr>
<tr>
<td>TBLWTH.B</td>
<td>$W_a, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{ROM}[W_a] = W_d$ for even ROM (byte)</td>
<td></td>
</tr>
<tr>
<td>TBLWTL</td>
<td>$W_a, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{ROM}[W_a] = W_d$ for even ROM (byte)</td>
<td></td>
</tr>
<tr>
<td>TBLWTL.B</td>
<td>$W_a, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{ROM}[W_a] = W_d$ for even ROM (byte)</td>
<td></td>
</tr>
<tr>
<td>ULNK</td>
<td>Deallocate Stack Frame</td>
<td></td>
</tr>
<tr>
<td>URUN</td>
<td>ICD Run</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>$W_a, W_b, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = W_a \oplus W_b$</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>$f, F$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$f = f \oplus W_0$</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>$f, W$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_0 = f \oplus W_0$</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>$W_a, \text{lit}5, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = W_a \oplus \text{lit}5$ (byte)</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>$\text{lit}10, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = W_d \oplus \text{lit}10$ (byte)</td>
<td></td>
</tr>
<tr>
<td>XOR.B</td>
<td>$\text{lit}10, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = W_d \oplus \text{lit}10$ (byte)</td>
<td></td>
</tr>
<tr>
<td>XOR.B</td>
<td>$f, W$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_0 = f \oplus W_0$ (byte)</td>
<td></td>
</tr>
<tr>
<td>XOR.B</td>
<td>$W_a, \text{lit}5, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = W_a \oplus \text{lit}5$ (byte)</td>
<td></td>
</tr>
<tr>
<td>XOR.B</td>
<td>$W_a, W_b, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = W_a \oplus W_b$ (byte)</td>
<td></td>
</tr>
<tr>
<td>XOR.B</td>
<td>$f, F$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$f = f \oplus W_0$ (byte)</td>
<td></td>
</tr>
<tr>
<td>ZE</td>
<td>$W_a, W_d$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W_d = W_a &amp; FF$</td>
<td></td>
</tr>
</tbody>
</table>

---

#bank_dma

**Syntax:**

```
#BANK_DMA
```

**Elements:** None

**Purpose:** Tells the compiler to assign the data for the next variable, array or structure into DMA bank

**Examples:**

```c
#bank_dma
struct {
    int r_w;
    int c_w;
    long unused :2;
    long data: 4;
} a_port;  //the data for a_port will be forced into memory
bank DMA
```
#bankx

**Syntax:**  
#BANKX

**Elements:**  
None

**Purpose:**  
Tells the compiler to assign the data for the next variable, array, or structure into Bank X.

**Examples:**
```c
#bankx
struct {
    int r_w;
    int c_d;
    long unused : 2;
    long data : 4;
} a_port;
// The data for a_port will be forced into memory bank x.
```

**Example Files:**  
None

**Also See:**  
None

---

#banky

**Syntax:**  
#BANKY

**Elements:**  
None

**Purpose:**  
Tells the compiler to assign the data for the next variable, array, or structure into Bank Y.

**Examples:**
```c
#banky
struct {
    int r_w;
    int c_d;
    long unused : 2;
```
long data : 4;
} a_port;
// The data for a_port will be forced into memory bank y.

Example Files: None

Also See: None

#bit

Syntax: 

#BIT id = x.y

Elements: 

id is a valid C identifier,
x is a constant or a C variable,
y is a constant 0-7 (for 8-bit PICs)
y is a constant 0-15 (for 16-bit PICs)

Purpose: 

A new C variable (one bit) is created and is placed in memory at byte x and bit y. This is useful to gain access in C directly to a bit in the processors special function register map. It may also be used to easily access a bit of a standard C variable.

Examples: 

#bit T1IF = 0x 84.3
...
T1IF = 0; // Clear Timer 0 interrupt flag

int result;
#bit result_odd = result.0
...
if (result_odd)

Example Files: ex_glint.c

Also See: #BYTE, #RESERVE, #LOCATE, #WORD
**__buildcount__**

Only defined if Options>Project Options>Global Defines has global defines enabled.

This id resolves to a number representing the number of successful builds of the project.

#build

**Syntax:**

- `#BUILD(segment = address)`
- `#BUILD(segment = address, segment = address)`
- `#BUILD(segment = start:end)`
- `#BUILD(segment = start: end, segment = start: end)`
- `#BUILD(nosleep)`
- `#BUILD(segment = size) : For STACK use only`
- `#BUILD(ALT_INTERRUPT)`
- `#BUILD(AUX_MEMORY)`

**Elements:**

- `segment` is one of the following memory segments which may be assigned a location: RESET, INTERRUPT, or STACK

  - `address` is a ROM location memory address. Start and end are used to specify a range in memory to be used. Start is the first ROM location and end is the last ROM location to be used.

  - `RESET` will move the compiler's reset vector to the specified location.
  - `INTERRUPT` will move the compiler's interrupt service routine to the specified location. This just changes the location the compiler puts it's reset and ISR, it doesn't change the actual vector of the PIC. If you specify a range that is larger than actually needed, the extra space will not be used and prevented from use by the compiler.

  - `STACK` configures the range (start and end locations) used for the stack, if not specified the compiler uses the last 256 bytes. The STACK can be specified by only using the size parameters. In this case, the compiler uses the last RAM locations on the chip and builds the stack below it.

  - `ALT_INTERRUPT` will move the compiler's interrupt service routine to the alternate location, and configure the PIC to use the alternate location.

  - `nosleep` is used to prevent the compiler from inserting a sleep at the end
of main()

*Bootload* produces a bootloader-friendly hex file (in order, full block size).

*NOSLEEP_LOCK* is used instead of a sleep at the end of a main infinite loop.

*AUX_MEMORY* - Only available on devices with an auxiliary memory segment. Causes compiler to build code for the auxiliary memory segment, including the auxiliary reset and interrupt vectors. Also enables the keyword *INT_AUX* which is used to create the auxiliary interrupt service routine.

**Purpose:** When linking multiple compilation units, this directive must appear exactly the same in each compilation unit.

These directives are commonly used in bootloaders, where the reset and interrupt needs to be moved to make space for the bootloading application.

**Examples:**

/* assign the location where the compiler will place the reset and interrupt vectors */
#define(reset=0x200,interrupt=0x208)

/* assign the location and fix the size of the segments used by the compiler for the reset and interrupt vectors */
#define(reset=0x200:0x207, interrupt=0x208:0x2ff)

/* assign stack space of 512 bytes */
#define(stack=0x1E00:0x1FFF)

#define(stack= 0x300) // When Start and End locations are not specified, the compiler uses the last RAM locations available on the chip.

**Example Files:** None

**Also See:** [#LOCATE, #RESERVE, ROM, ORG]
#byte

Syntax: #byte \textit{id} = x

Elements: \textit{id} is a valid C identifier, \textit{x} is a C variable or a constant

Purpose: If the \textit{id} is already known as a C variable then this will locate the variable at address \textit{x}. In this case the variable type does not change from the original definition. If the \textit{id} is not known a new C variable is created and placed at address \textit{x} with the type int (8 bit)

Warning: In both cases memory at \textit{x} is not exclusive to this variable. Other variables may be located at the same location. In fact when \textit{x} is a variable, then \textit{id} and \textit{x} share the same memory location.

Examples:

\begin{verbatim}
#byte status_register = 0x42
#byte b_port = 0x02C8

struct {
    short int r_w;
    short int c_d;

    int data : 6 ; } E_port;
#byte a_port = 0x2DA
...
a_port.c_d = 1;
\end{verbatim}

Example Files: ex_glint.c

Also See: #bit, #locate, #reserve, #word, Named Registers, Type Specifiers, Type Qualifiers, Enumerated Types, Structures & Unions, Typedef

#case

Syntax: #CASE

Elements: None

Purpose: Will cause the compiler to be case sensitive. By default the compiler is case insensitive. When linking multiple compilation units, this directive
must appear exactly the same in each compilation unit.

Warning: Not all the CCS example programs, headers and drivers have been tested with case sensitivity turned on.

**Examples:**

```
#include

int STATUS;

void func() {
    int status;
    ...
    STATUS = status; // Copy local status to
    // global
}
```

**Example Files:**  `ex_cust.c`

**Also See:** None

---

**_date_**

**Syntax:** `__DATE__`

**Elements:** None

**Purpose:** This pre-processor identifier is replaced at compile time with the date of the compile in the form: "31-JAN-03"

**Examples:**

```c
printf("Software was compiled on ");
printf(__DATE__);  
```

**Example Files:** None

**Also See:** None
#define

Syntax:

```
#define id text
or
#define id(x,y...) text
```

Elements:

- `id` is a preprocessor identifier, `text` is any text, `x,y` is a list of local preprocessor identifiers, and in this form there may be one or more identifiers separated by commas.

Purpose:

Used to provide a simple string replacement of the ID with the given text from this point of the program and on.

In the second form (a C macro) the local identifiers are matched up with similar identifiers in the text and they are replaced with text passed to the macro where it is used.

If the text contains a string of the form `#idx` then the result upon evaluation will be the parameter `id` concatenated with the string `x`.

If the text contains a string of the form `#idx#idy` then parameter `idx` is concatenated with parameter `idy` forming a new identifier.

Within the define text two special operators are supported:

- `#x` is the stringize operator resulting in "x"
- `x##y` is the concatenation operator resulting in `xy`

The varadic macro syntax is supported where the last parameter is specified as `...` and the local identifier used is `__va_args__`. In this case, all remaining arguments are combined with the commas.

Examples:

```
#define BITS 8
a=a+BITS;  //same as  a=a+8;

#define hi(x) (x<<4)
a=hi(a);  //same as  a=(a<<4);

#define isequal(a,b)  (primary_##a[b]==backup_##a[b])
    // usage isequal(names,5) is the same as
    // (primary_names[5]==backup_names[5])

#define str(s) #s
#define part(device)  #include str(device##.h)
    // usage part(16F887) is the same as
```
// #include "16F887.h"

#define DBG(...) fprintf(debug,__VA_ARGS__)

Example Files:  
ex_stwt.c,  
ex_macro.c

Also See:  
#UNDEF,  
#IFDEF,  
#IFNDEF

definedinc

Syntax:  

value = definedinc( variable );

Parameters:  
variable is the name of the variable, function, or type to be checked.

Returns:  
A C status for the type of id entered as follows:
0 – not known
1 – typedef or enum
2 – struct or union type
3 – typemod qualifier
4 – defined function
5 – function prototype
6 – compiler built-in function
7 – local variable
8 – global variable

Function:  
This function checks the type of the variable or function being passed in and returns a specific C status based on the type.

Availability:  
All devices

Requires:  
None.

Examples:  
int x, y = 0;
y = definedinc( x );  // y will return 7 – x is a local variable

Example Files:  
None

Also See:  
None
#device

**Syntax:**

```cpp
#DEVICE chip options
#DEVICE Compilation mode selection
```

**Elements:**

### Chip Options

*chip* is the name of a specific processor (like: `dsPIC33FJ64GP306`), To get a current list of supported devices:

```
START | RUN | CCSC +Q
```

*Options* are qualifiers to the standard operation of the device. Valid options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC=x</td>
<td>Where x is the number of bits read_adc() should return</td>
</tr>
<tr>
<td>ADC=SIGNED</td>
<td>Result returned from read_adc() is signed. (Default is unsigned)</td>
</tr>
<tr>
<td>ADC=UNSIGNED</td>
<td>Return result from read_adc() is unsigned. (Default is UNSIGNED)</td>
</tr>
<tr>
<td>ICD=TRUE</td>
<td>Generates code compatible with Microchips ICD debugging hardware.</td>
</tr>
<tr>
<td>ICD=n</td>
<td>For chips with multiple ICSP ports specify the port number being used. The default is 1.</td>
</tr>
<tr>
<td>WRITE_EEPROM=ASYNC</td>
<td>Prevents WRITE_EEPROM from hanging while writing is taking place. When used, do not write to EEPROM from both ISR and outside ISR.</td>
</tr>
<tr>
<td>WRITE_EEPROM = NOINT</td>
<td>Allows interrupts to occur while the write_eeprom() operations is polling the done bit to check if the write operations has completed. Can be used as long as no EEPROM operations are performed during an ISR.</td>
</tr>
<tr>
<td>HIGH_INTS=TRUE</td>
<td>Use this option for high/low priority interrupts on the PIC® 18.</td>
</tr>
<tr>
<td>%f=.</td>
<td>No 0 before a decimal point on %f numbers less than 1.</td>
</tr>
<tr>
<td>OVERLOAD=KEYWORD</td>
<td>Overloading of functions is now</td>
</tr>
<tr>
<td>PreProcessor</td>
<td>OVERLOAD=AUTO</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>PASS_STRINGS=IN_RAM</td>
<td>A new way to pass constant strings to a function by first copying the string to RAM and then passing a pointer to RAM to the function.</td>
</tr>
<tr>
<td>CONST=READ_ONLY</td>
<td>Uses the ANSI keyword CONST definition, making CONST variables read only, rather than located in program memory.</td>
</tr>
<tr>
<td>CONST=ROM</td>
<td>Uses the CCS compiler traditional keyword CONST definition, making CONST variables located in program memory.</td>
</tr>
<tr>
<td>NESTED_INTERRUPTS=TRUE</td>
<td>Enables interrupt nesting for PIC24, dsPIC30, and dsPIC33 devices. Allows higher priority interrupts to interrupt lower priority interrupts.</td>
</tr>
<tr>
<td>NORETFIE</td>
<td>ISR functions (preceded by a #int_xxx) will use a RETURN opcode instead of the RETFIE opcode. This is not a commonly used option; used rarely in cases where the user is writing their own ISR handler.</td>
</tr>
<tr>
<td>NO_DIGITAL_INIT</td>
<td>Normally the compiler sets all I/O pins to digital and turns off the comparator. This option prevents that action.</td>
</tr>
<tr>
<td>DUAL_PARTITION</td>
<td>For devices with Dual Partition Flash Modes, this enables Dual Partition Flash mode by setting the FBOOT configuration register to the appropriate value. It cuts the available program memory in half, and moves the configuration register addresses to the Dual Partition locations.</td>
</tr>
<tr>
<td>DUAL_PARTITION_PROTECTED</td>
<td>For devices with Dual Partition Flash Modes this enabled Protected Dual Partition Flash mode, Partition 1 is write-protected when inactive, by setting the FBOOT configuration register to the appropriate value. It cuts the available program memory in half and moves the configuration</td>
</tr>
</tbody>
</table>
register addresses to the Dual Partition locations.

| PARTITION_SEQUENCE=x | A value from 0 to 4095 to set the FBTSEQ configuration register. Only used when either DUAL_PARTITION or DUAL_PARTITION_PROTECTED is used. The value is used to determine which partition is active on power-up. The Partition with the lowest value will be the active partition. If the value is the same for both partitions, then Partition 1 will be the active partition on power-up. |

Both chip and options are optional, so multiple #DEVICE lines may be used to fully define the device. Be warned that a #DEVICE with a chip identifier, will clear all previous #DEVICE and #FUSE settings.

**Compilation mode selection**
The #DEVICE directive supports compilation mode selection. The valid keywords are CCS2, CCS3, CCS4 and ANSI. The default mode is CCS4. For the CCS4 and ANSI mode, the compiler uses the default fuse settings NOLVP, PUT for chips with these fuses. The NOWDT fuse is default if no call is made to restart_wdt().

| CCS4 | This is the default compilation mode. |
| ANSI | Default data type is SIGNED all other modes default is UNSIGNED. Compilation is case sensitive, all other modes are case insensitive. |
| CCS2 CCS3 | var16 = NegConst8 is compiled as: var16 = NegConst8 & 0xff (no sign extension). The overload keyword is required. |
| CCS2 only | The default #DEVICE ADC is set to the resolution of the part, all other modes default to 8. onebit = eightbits is compiled as onebit = (eightbits != 0) All other modes compile as: onebit = (eightbits & 1) |

**Purpose:** *Chip Options* - Defines the target processor. Every program must have exactly one #DEVICE with a chip. When linking multiple compilation units, this directive must appear exactly the same in each compilation.
unit.

**Compilation mode selection** - The compilation mode selection allows existing code to be compiled without encountering errors created by compiler compliance. As CCS discovers discrepancies in the way expressions are evaluated according to ANSI, the change will generally be made only to the ANSI mode and the next major CCS release.

**Examples:**

**Chip Options**-

```plaintext
#device DSPIC33FJ64GP306
#device PIC24FJ64GA002 ICD=TRUE
#device ADC=10
#device ICD=TRUE ADC=10
```

**Float Options**-

```plaintext
#device %f=
printf("%f",.5); //will print .5, without the directive it will print 0.5
```

**Compilation mode selection**-

```plaintext
#device CCS2
```

**Example Files:** None

**Also See:** None

---

**_device_**

**Syntax:** __DEVICE__

**Elements:** None

**Purpose:** This pre-processor identifier is defined by the compiler with the base number of the current device (from a #DEVICE). The base number is usually the number after the C in the part number. For example the PIC16C622 has a base number of 622.

**Examples:**

```plaintext
#if __device__==71
SETUP_ADC_PORTS( ALL_DIGITAL );
#endif
```

**Example Files:** None

**Also See:** #DEVICE
### #if expr  #else  #elif  #endif

**Syntax:**

```plaintext
#if expr
code
#elif expr   //Optional, any number may be used
code
#else       //Optional
code
#endif
```

**Elements:**

- `expr` is an expression with constants, standard operators and/or preprocessor identifiers.
- `Code` is any standard C source code.

**Purpose:**

The pre-processor evaluates the constant expression and if it is non-zero will process the lines up to the optional `#ELSE` or the `#ENDIF`.

Note: you may NOT use C variables in the `#IF`. Only preprocessor identifiers created via `#define` can be used.

The preprocessor expression `DEFINED(id)` may be used to return 1 if the id is defined and 0 if it is not.

`==` and `!=` operators now accept a constant string as both operands. This allows for compile time comparisons and can be used with `GETENV()` when it returns a string result.

**Examples:**

```c
#if MAX_VALUE > 255
  long value;
#else
  int value;
#endif
#if getenv("DEVICE")=="PIC16F877"
  //do something special for the PIC16F877
#endif
```

**Example Files:**

`ex_extee.c`

**Also See:**

`#IFDEF, #IFNDEF, getenv()`
### #error

**Syntax:**

- `#ERROR text`
- `#ERROR / warning text`
- `#ERROR / information text`

**Elements:**

- `text` is optional and may be any text

**Purpose:**

Forces the compiler to generate an error at the location this directive appears in the file. The text may include macros that will be expanded for the display. This may be used to see the macro expansion. The command may also be used to alert the user to an invalid compile time situation.

**Examples:**

```c
#if BUFFER_SIZE>16
#error Buffer size is too large
#endif
#error Macro test: min(x,y)
```

**Example Files:**

- `ex_psp.c`

**Also See:**

- `#WARNING`

### #export (options)

**Syntax:**

- `#EXPORT (options)`

**Elements:**

- `FILE=filename`
  
  The filename which will be generated upon compile. If not given, the filename will be the name of the file you are compiling, with a `.o` or `.hex` extension (depending on output format).

- `ONLY=symbol+symbol+.....+symbol`
  
  Only the listed symbols will be visible to modules that import or link this relocatable object file. If neither `ONLY` or `EXCEPT` is used, all symbols are exported.

- `EXCEPT=symbol+symbol+.....+symbol`
  
  All symbols except the listed symbols will be visible to modules that import or link this relocatable object file. If neither `ONLY` or `EXCEPT` is used, all symbols are exported.
**RELOCATABLE**
CCS relocatable object file format. Must be imported or linked before loading into a PIC. This is the default format when the #EXPORT is used.

**HEX**
Intel HEX file format. Ready to be loaded into a PIC. This is the default format when no #EXPORT is used.

**RANGE=start:stop**
Only addresses in this range are included in the hex file.

**OFFSET=address**
Hex file address starts at this address (0 by default)

**ODD**
Only odd bytes place in hex file.

**EVEN**
Only even bytes placed in hex file.

**Purpose:**
This directive will tell the compiler to either generate a relocatable object file or a stand-alone HEX binary. A relocatable object file must be linked into your application, while a stand-alone HEX binary can be programmed directly into the PIC.
The command line compiler and the PCW IDE Project Manager can also be used to compile/link/build modules and/or projects.
Multiple #EXPORT directives may be used to generate multiple hex files. this may be used for 8722 like devices with external memory.

**Examples:**
```c
#EXPORT(RELOCATABLE, ONLY=TimerTask)
void TimerFunc1(void) { /* some code */ }
void TimerFunc2(void) { /* some code */ }
void TimerFunc3(void) { /* some code */ }
void TimerTask(void)
{
    TimerFunc1();
    TimerFunc2();
    TimerFunc3();
}
/*
This source will be compiled into a relocatable object, but the object this is being linked to can only see TimerTask() */
```
### __file__

**Syntax:**  
__FILE__

**Elements:** None

**Purpose:** The pre-processor identifier is replaced at compile time with the file path and the filename of the file being compiled.

**Examples:**
```c
if(index>MAX_ENTRIES)
    printf("Too many entries, source file: "
            __FILE__ " at line " __LINE__ "\r\n");
```

**Example Files:** assert.h

**Also See:** __line__

### __filename__

**Syntax:**  
__FILENAME__

**Elements:** None

**Purpose:** The pre-processor identifier is replaced at compile time with the filename of the file being compiled.

**Examples:**
```c
if(index>MAX_ENTRIES)
    printf("Too many entries, source file: "
            __FILENAME__ " at line " __LINE__ "\r\n");
```

**Example Files:** None

**Also See:** __line__
#fill_rom

**Syntax:**

```
#fill_rom value
```

**Elements:**

`value` is a constant 16-bit value

**Purpose:**

This directive specifies the data to be used to fill unused ROM locations. When linking multiple compilation units, this directive must appear exactly the same in each compilation unit.

**Examples:**

```
#fill_rom 0x36
```

**Example Files:**
None

**Also See:**

#ROM

---

#fuses

**Syntax:**

```
#FUSES options
```

**Elements:**

`options` vary depending on the device. A list of all valid options has been put at the top of each device .h file in a comment for reference. The PCW device edit utility can modify a particular device’s fuses. The PCW pull down menu VIEW | Valid fuses will show all fuses with their descriptions.

Some common options are:
- LP, XT, HS, RC
- WDT, NOWDT
- PROTECT, NOPROTECT
- PUT, NOPUT (Power Up Timer)
- BROWNOUT, NOBROWNOUT

**Purpose:**

This directive defines what fuses should be set in the part when it is programmed. This directive does not affect the compilation; however, the information is put in the output files. If the fuses need to be in Parallax
format, add a PAR option. SWAP has the special function of swapping (from the Microchip standard) the high and low BYTES of non-program data in the Hex file. This is required for some device programmers.

Some fuses are set by the compiler based on other compiler directives. For example, the oscillator fuses are set up by the #USE delay directive. The debug, No debug and ICSPN Fuses are set by the #DEVICE ICD=directive.

Some processors allow different levels for certain fuses. To access these levels, assign a value to the fuse.

When linking multiple compilation units be aware this directive applies to the final object file. Later files in the import list may reverse settings in previous files.

To eliminate all fuses in the output files use:

```
#FUSES none
```

To manually set the fuses in the output files use:

```
#FUSES 1 = 0xC200 // sets config word 1 to 0xC200
```

**Examples:**

```
#fuses HS,NOWDT
```

**Example Files:** None

**Also See:** None

---

### `#hexcomment`

**Syntax:**

```
#HEXCOMMENT text comment for the top of the hex file
#HEXCOMMENT\ text comment for the end of the hex file
```

**Elements:** None

**Purpose:** Puts a comment in the hex file

Some programmers (MPLAB in particular) do not like comments at the top of the hex file.
#id

## Syntax:

- `#ID number 32`
- `#ID number, number, number, number`
- `#ID "filename"`
- `#ID CHECKSUM`

## Elements:

- **Number 32**: is a 32 bit number,
- **number**: is a 8 bit number,
- **filename**: is any valid PC filename and **checksum** is a keyword.

## Purpose:

This directive defines the ID word to be programmed into the part. This directive does not affect the compilation but the information is put in the output file.

The first syntax will take a 32-bit number and put one byte in each of the four ID bytes in the traditional manner. The second syntax specifies the exact value to be used in each of the four ID bytes.

When a filename is specified the ID is read from the file. The format must be simple text with a CR/LF at the end. The keyword CHECKSUM indicates the device checksum should be saved as the ID.

## Examples:

- `#id 0x12345678`
- `#id 0x12, 0x34, 0x45, 0x67`
- `#id "serial.num"`
- `#id CHECKSUM`

## Example Files:

- `ex_cust.c`

## Also See:

- None
### #if expr #else #elif #endif

**Syntax:**

```
#if expr
code
#else
code
#if expr
//Optional, any number may be used
code
#elif expr
//Optional
code
#endif
```

**Elements:**

- `expr` is an expression with constants, standard operators and/or preprocessor identifiers.
- `Code` is any standard C source code.

**Purpose:**

The pre-processor evaluates the constant expression and if it is non-zero will process the lines up to the optional `#ELSE` or the `#ENDIF`.

Note: you may NOT use C variables in the `#IF`. Only preprocessor identifiers created via `#define` can be used.

The preprocessor expression `DEFINED(id)` may be used to return 1 if the id is defined and 0 if it is not.

`==` and `!=` operators now accept a constant string as both operands. This allows for compile time comparisons and can be used with `GETENV()` when it returns a string result.

**Examples:**

```
#if MAX_VALUE > 255
  long value;
#else
  int value;
#endif
#if getenv("DEVICE")=="PIC16F877"
  //do something special for the PIC16F877
#endif
```

**Example Files:**

- `ex_extee.c`

**Also See:**

- `#IFDEF`, `#IFNDEF`, `getenv()`
### #ifdef  #ifndef  #else  #elif  #endif

**Syntax:**

```
#IFDEF id
code
#ELIF code
#ELSE code
#ENDIF

#IFNDEF id
code
#ELIF code
#ELSE code
#ENDIF
```

**Elements:**

- **id** is a preprocessor identifier,
- **code** is valid C source code.

**Purpose:**

This directive acts much like the #IF except that the preprocessor simply checks to see if the specified ID is known to the preprocessor (created with a #DEFINE). #IFDEF checks to see if defined and #IFNDEF checks to see if it is not defined.

**Examples:**

```c
#define debug    // Comment line out for no debug
...
 ifdef DEBUG
 printf("debug point a");
 endif
```

**Example Files:**

- ex_sqw.c

**Also See:**

- #IF

---

### #ignore_warnings

**Syntax:**

```
#define ignore_warnings ALL
#define IGNORE_WARNINGS NONE
#define IGNORE_WARNINGS warnings
```

---

132
**PreProcessor**

<table>
<thead>
<tr>
<th>Elements:</th>
<th><strong>warnings</strong> is one or more warning numbers separated by commas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>This function will suppress warning messages from the compiler. ALL indicates no warning will be generated. NONE indicates all warnings will be generated. If numbers are listed then those warnings are suppressed.</td>
</tr>
</tbody>
</table>
| Examples: | #ignore_warnings 203  
while(TRUE) {  
#ignore_warnings NONE |
| Example Files: | None |
| Also See: | Warning messages |

## #import (options)

### Syntax:

```
#IMPORT (options)
```

### Elements:

**FILE=filename**  
The filename of the object you want to link with this compilation.

**ONLY=symbol+symbol+.....+symbol**  
Only the listed symbols will imported from the specified relocatable object file. If neither ONLY or EXCEPT is used, all symbols are imported.

**EXCEPT=symbol+symbol+.....+symbol**  
The listed symbols will not be imported from the specified relocatable object file. If neither ONLY or EXCEPT is used, all symbols are imported.

**RELOCATABLE**  
CCS relocatable object file format. This is the default format when the #IMPORT is used.

**COFF**  
COFF file format from MPASM, C18 or C30.

**HEX**  
Imported data is straight hex data.
### RANGE=start:stop
Only addresses in this range are read from the hex file.

### LOCATION=id
The identifier is made a constant with the start address of the imported data.

### SIZE=id
The identifier is made a constant with the size of the imported data.

**Purpose:** This directive will tell the compiler to include (link) a relocatable object with this unit during compilation. Normally all global symbols from the specified file will be linked, but the EXCEPT and ONLY options can prevent certain symbols from being linked.

The command line compiler and the PCW IDE Project Manager can also be used to compile/link/build modules and/or projects.

**Examples:**
```c
#include <filename>
or
#include "filename"
```

**Elements:**
- `filename` is a valid PC filename. It may include normal drive and path information. A file with the extension ".encrypted" is a valid PC file. The standard compiler `#INCLUDE` directive will accept files with this extension.

---

**Examples:**
```c
#include <filename>
or
#include "filename"
```

**Elements:**
- `filename` is a valid PC filename. It may include normal drive and path information. A file with the extension ".encrypted" is a valid PC file. The standard compiler `#INCLUDE` directive will accept files with this extension.

---

**Examples:**
```c
#include <filename>
or
#include "filename"
```

**Elements:**
- `filename` is a valid PC filename. It may include normal drive and path information. A file with the extension ".encrypted" is a valid PC file. The standard compiler `#INCLUDE` directive will accept files with this extension.
extension and decrypt them as they are read. This allows include files to be distributed without releasing the source code.

**Purpose:** Text from the specified file is used at this point of the compilation. If a full path is not specified the compiler will use the list of directories specified for the project to search for the file. If the filename is in "" then the directory with the main source file is searched first. If the filename is in <> then the directory with the main source file is searched last.

**Examples:**

```c
#include <16C54.H>
#include <C:\INCLUDES\COMLIB\MYRS232.C>
```

**Example Files:** ex_sqw.c

**Also See:** None

### #inline

**Syntax:**

```
#INLINE
```

**Elements:** None

**Purpose:** Tells the compiler that the function immediately following the directive is to be implemented INLINE. This will cause a duplicate copy of the code to be placed everywhere the function is called. This is useful to save stack space and to increase speed. Without this directive the compiler will decide when it is best to make procedures INLINE.

**Examples:**

```c
#include
swapbyte(int &a, int &b) {
  int t;
  t=a;
  a=b;
  b=t;
}
```

**Example Files:** ex_cust.c

**Also See:** #SEPARATE
#int_xxxx

## Syntax:

<table>
<thead>
<tr>
<th>#INT_AC1</th>
<th>Analog comparator 1 output change</th>
</tr>
</thead>
<tbody>
<tr>
<td>#INT_AC2</td>
<td>Analog comparator 2 output change</td>
</tr>
<tr>
<td>#INT_AC3</td>
<td>Analog comparator 3 output change</td>
</tr>
<tr>
<td>#INT_AC4</td>
<td>Analog comparator 4 output change</td>
</tr>
<tr>
<td>#INT_ADC1</td>
<td>ADC1 conversion complete</td>
</tr>
<tr>
<td>#INT_ADC2</td>
<td>Analog to digital conversion complete</td>
</tr>
<tr>
<td>#INT_ADCP0</td>
<td>ADC pair 0 conversion complete</td>
</tr>
<tr>
<td>#INT_ADCP1</td>
<td>ADC pair 1 conversion complete</td>
</tr>
<tr>
<td>#INT_ADCP2</td>
<td>ADC pair 2 conversion complete</td>
</tr>
<tr>
<td>#INT_ADCP3</td>
<td>ADC pair 3 conversion complete</td>
</tr>
<tr>
<td>#INT_ADCP4</td>
<td>ADC pair 4 conversion complete</td>
</tr>
<tr>
<td>#INT_ADCP5</td>
<td>ADC pair 5 conversion complete</td>
</tr>
<tr>
<td>#INT_ADDRERR</td>
<td>Address error trap</td>
</tr>
<tr>
<td>#INT_C1RX</td>
<td>ECAN1 Receive Data Ready</td>
</tr>
<tr>
<td>#INT_C1TX</td>
<td>ECAN1 Transmit Data Request</td>
</tr>
<tr>
<td>#INT_C2RX</td>
<td>ECAN2 Receive Data Ready</td>
</tr>
<tr>
<td>#INT_C2TX</td>
<td>ECAN2 Transmit Data Request</td>
</tr>
<tr>
<td>#INT_CAN1</td>
<td>CAN 1 Combined Interrupt Request</td>
</tr>
<tr>
<td>#INT_CAN2</td>
<td>CAN 2 Combined Interrupt Request</td>
</tr>
<tr>
<td>#INT_CNI</td>
<td>Input change notification interrupt</td>
</tr>
<tr>
<td>#INT_COMP</td>
<td>Comparator event</td>
</tr>
<tr>
<td>#INT_CRC</td>
<td>Cyclic redundancy check generator</td>
</tr>
<tr>
<td>#INT_DC1</td>
<td>DCI transfer done</td>
</tr>
<tr>
<td>#INT_DCIE</td>
<td>DCE error</td>
</tr>
<tr>
<td>#INT_DMA0</td>
<td>DMA channel 0 transfer complete</td>
</tr>
<tr>
<td>#INT_DMA1</td>
<td>DMA channel 1 transfer complete</td>
</tr>
<tr>
<td>#INT_DMA2</td>
<td>DMA channel 2 transfer complete</td>
</tr>
<tr>
<td>#INT_DMA3</td>
<td>DMA channel 3 transfer complete</td>
</tr>
<tr>
<td>#INT_DMA4</td>
<td>DMA channel 4 transfer complete</td>
</tr>
<tr>
<td>#INT_DMA5</td>
<td>DMA channel 5 transfer complete</td>
</tr>
<tr>
<td>#INT_DMA6</td>
<td>DMA channel 6 transfer complete</td>
</tr>
<tr>
<td>#INT_DMA7</td>
<td>DMA channel 7 transfer complete</td>
</tr>
<tr>
<td>#INT_DMAERR</td>
<td>DMAC error trap</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>#INT_EEPROM</td>
<td>Write complete</td>
</tr>
<tr>
<td>#INT_EX1</td>
<td>External Interrupt 1</td>
</tr>
<tr>
<td>#INT_EX4</td>
<td>External Interrupt 4</td>
</tr>
<tr>
<td>#INT_EXT0</td>
<td>External Interrupt 0</td>
</tr>
<tr>
<td>#INT_EXT1</td>
<td>External interrupt #1</td>
</tr>
<tr>
<td>#INT_EXT2</td>
<td>External interrupt #2</td>
</tr>
<tr>
<td>#INT_EXT3</td>
<td>External interrupt #3</td>
</tr>
<tr>
<td>#INT_EXT4</td>
<td>External interrupt #4</td>
</tr>
<tr>
<td>#INT_FAULTA</td>
<td>PWM Fault A</td>
</tr>
<tr>
<td>#INT_FAULTA2</td>
<td>PWM Fault A 2</td>
</tr>
<tr>
<td>#INT_FAULTB</td>
<td>PWM Fault B</td>
</tr>
<tr>
<td>#INT_IC1</td>
<td>Input Capture #1</td>
</tr>
<tr>
<td>#INT_IC2</td>
<td>Input Capture #2</td>
</tr>
<tr>
<td>#INT_IC3</td>
<td>Input Capture #3</td>
</tr>
<tr>
<td>#INT_IC4</td>
<td>Input Capture #4</td>
</tr>
<tr>
<td>#INT_IC5</td>
<td>Input Capture #5</td>
</tr>
<tr>
<td>#INT_IC6</td>
<td>Input Capture #6</td>
</tr>
<tr>
<td>#INT_IC7</td>
<td>Input Capture #7</td>
</tr>
<tr>
<td>#INT_IC8</td>
<td>Input Capture #8</td>
</tr>
<tr>
<td>#INT_LOWVOLT</td>
<td>Low voltage detected</td>
</tr>
<tr>
<td>#INT_LVD</td>
<td>Low voltage detected</td>
</tr>
<tr>
<td>#INT_MATHERR</td>
<td>Arithmetic error trap</td>
</tr>
<tr>
<td>#INT_MI2C</td>
<td>Master I2C activity</td>
</tr>
<tr>
<td>#INT_MI2C2</td>
<td>Master2 I2C activity</td>
</tr>
<tr>
<td>#INT_OC1</td>
<td>Output Compare #1</td>
</tr>
<tr>
<td>#INT_OC2</td>
<td>Output Compare #2</td>
</tr>
<tr>
<td>#INT_OC3</td>
<td>Output Compare #3</td>
</tr>
<tr>
<td>#INT_OC4</td>
<td>Output Compare #4</td>
</tr>
<tr>
<td>#INT_OC5</td>
<td>Output Compare #5</td>
</tr>
<tr>
<td>#INT_OC6</td>
<td>Output Compare #6</td>
</tr>
<tr>
<td>#INT_OC7</td>
<td>Output Compare #7</td>
</tr>
<tr>
<td>#INT_OC8</td>
<td>Output Compare #8</td>
</tr>
<tr>
<td>#INT_OSC_FAIL</td>
<td>System oscillator failed</td>
</tr>
<tr>
<td>#INT_PMP</td>
<td>Parallel master port</td>
</tr>
<tr>
<td>#INT_PMP2</td>
<td>Parallel master port 2</td>
</tr>
<tr>
<td>#INT_PWM1</td>
<td>PWM generator 1 time based interrupt</td>
</tr>
<tr>
<td>Directive</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>#INT_PWM2</td>
<td>PWM generator 2 time based interrupt</td>
</tr>
<tr>
<td>#INT_PWM3</td>
<td>PWM generator 3 time based interrupt</td>
</tr>
<tr>
<td>#INT_PWM4</td>
<td>PWM generator 4 time based interrupt</td>
</tr>
<tr>
<td>#INT_PWMSEM</td>
<td>PWM special event trigger</td>
</tr>
<tr>
<td>#INT_QEI</td>
<td>QEI position counter compare</td>
</tr>
<tr>
<td>#INT_RDA</td>
<td>RS232 receive data available</td>
</tr>
<tr>
<td>#INT_RDA2</td>
<td>RS232 receive data available in buffer 2</td>
</tr>
<tr>
<td>#INT_RTC</td>
<td>Real - Time Clock/Calendar</td>
</tr>
<tr>
<td>#INT_SI2C</td>
<td>Slave I2C activity</td>
</tr>
<tr>
<td>#INT_SI2C2</td>
<td>Slave2 I2C activity</td>
</tr>
<tr>
<td>#INT_SPI1</td>
<td>SPI1 Transfer Done</td>
</tr>
<tr>
<td>#INT_SPI1E</td>
<td>SPI1E Transfer Done</td>
</tr>
<tr>
<td>#INT_SPI2</td>
<td>SPI2 Transfer Done</td>
</tr>
<tr>
<td>#INT_SPI2E</td>
<td>SPI2 Error</td>
</tr>
<tr>
<td>#INT_SPIE</td>
<td>SPI Error</td>
</tr>
<tr>
<td>#INT_STACKERR</td>
<td>Stack Error</td>
</tr>
<tr>
<td>#INT_TBE</td>
<td>RS232 transmit buffer empty</td>
</tr>
<tr>
<td>#INT_TBE2</td>
<td>RS232 transmit buffer 2 empty</td>
</tr>
<tr>
<td>#INT_TIMER1</td>
<td>Timer 1 overflow</td>
</tr>
<tr>
<td>#INT_TIMER2</td>
<td>Timer 2 overflow</td>
</tr>
<tr>
<td>#INT_TIMER3</td>
<td>Timer 3 overflow</td>
</tr>
<tr>
<td>#INT_TIMER4</td>
<td>Timer 4 overflow</td>
</tr>
<tr>
<td>#INT_TIMER5</td>
<td>Timer 5 overflow</td>
</tr>
<tr>
<td>#INT_TIMER6</td>
<td>Timer 6 overflow</td>
</tr>
<tr>
<td>#INT_TIMER7</td>
<td>Timer 7 overflow</td>
</tr>
<tr>
<td>#INT_TIMER8</td>
<td>Timer 8 overflow</td>
</tr>
<tr>
<td>#INT_TIMER9</td>
<td>Timer 9 overflow</td>
</tr>
<tr>
<td>#INT_UART1E</td>
<td>UART1 error</td>
</tr>
<tr>
<td>#INT_UART2E</td>
<td>UART2 error</td>
</tr>
<tr>
<td>#INT_AUX</td>
<td>Auxiliary memory ISR</td>
</tr>
</tbody>
</table>

**Elements:** NOCLEAR, LEVEL=n, HIGH, FAST, ALT

**Purpose:** These directives specify the following function is an interrupt function. Interrupt functions may not have any parameters. Not all directives may be used with all parts. See the devices .h file for all valid interrupts for the part or in PCW use the pull down VIEW | Valid Ints
The MPU will jump to the function when the interrupt is detected. The compiler will generate code to save and restore the machine state, and will clear the interrupt flag. To prevent the flag from being cleared add NOCLEAR after the #INT_xxxx. The application program must call ENABLE_INTERRUPTS(INT_xxxx) to initially activate the interrupt.

An interrupt marked FAST uses the shadow feature to save registers. Only one interrupt may be marked fast. Any registers used in the FAST interrupt beyond the shadow registers is the responsibility of the user to save and restore.

Level=n specifies the level of the interrupt. Higher numbers are a higher priority.

Enable_interrupts specifies the levels that are enabled. The default is level 0 and level 7 is never disabled.

High is the same as level = 7.

A summary of the different kinds of dsPIC/PIC24 interrupts:

- **#INT_xxxx**
  - Normal (low priority) interrupt. Compiler saves/restores key registers.
  - This interrupt will not interrupt any interrupt in progress.

- **#INT_xxxx FAST**
  - Compiler does a FAST save/restore of key registers.
  - Only one is allowed in a program.

- **#INT_xxxxLevel=3**
  - Interrupt is enabled when levels 3 and below are enabled.

- **#INT_GLOBAL**
  - Compiler generates no interrupt code. User function is located at address 8 for user interrupt handling.

- **#INT_xxxx ALT**
  - Interrupt is placed in Alternate Interrupt Vector instead of Default Interrupt Vector.

**Examples:**

```c
#int_ad
adc_handler() {
    adc_active=FALSE;
}
```
**#int_timer1 noclear**

```c
isr() {
    ...
}
```

**Example Files:** None

**Also See:** enable_interrupts(), disable_interrupts(), #INT_DEFAULT.

---

## #INT_DEFAULT

**Syntax:** #INT_DEFAULT

**Elements:** None

**Purpose:** The following function will be called if the ds PIC® triggers an interrupt and a #INT_xxx handler has not been defined for the interrupt.

**Examples:**

```c
#define #INT_DEFAULT
default_isr() {
    printf("Unexplained interrupt\r\n");
}
```

**Example Files:** None

**Also See:** #INT_xxxx,

---

## __line__

**Syntax:** __line__

**Elements:** None

**Purpose:** The pre-processor identifier is replaced at compile time with line number of the file being compiled.

**Examples:**

```c
if(index>MAX_ENTRIES)
```
```c
printf("Too many entries, source file: "
"%FILE%" at line " %LINE% \
\n");
```

**Example Files:**  
assert.h

**Also See:**  
/__file__

### #list

**Syntax:**  
#LIST

**Elements:**  
None

**Purpose:**  
#LIST begins inserting or resumes inserting source lines into the .LST file after a #NOLIST.

**Examples:**  
#NOLIST // Don't clutter up the list file  
#include <cdriver.h>  
#LIST

**Example Files:**  
16c74.h

**Also See:**  
#NOLIST

### #line

**Syntax:**  
#LINE number file name

**Elements:**  
Number is non-negative decimal integer. File name is optional.

**Purpose:**  
The C pre-processor informs the C Compiler of the location in your source code. This code is simply used to change the value of _LINE_ and _FILE_ variables.

**Examples:**  
1. void main(){  
    #line 10  // specifies the line number that  
    // should be reported for  
    // the following line of input
```
2. #line 7 "hello.c"
   // line number in the source file
   // hello.c and it sets the
   // line 7 as current line
   // and hello.c as current file

Example Files:  None

Also See:  None

#locate

Syntax:  #LOCATE id=x

Elements:  id is a C variable,
           x is a constant memory address

Purpose:  #LOCATE allocates a C variable to a specified address. If the C variable
          was not previously defined, it will be defined as an INT8.

          A special form of this directive may be used to locate all A functions local
          variables starting at a fixed location.
          Use: #LOCATE Auto = address

          This directive will place the indirlected C variable at the requested
          address.

Examples:  // This will locate the float variable at 50-53
           // and C will not use this memory for other
           // variables automatically located.
           float x;
           #locate x=0x800

Example Files:  ex_glint.c

Also See:  #byte, #bit, #reserve, #word, Named Registers, Type Specifiers, Type
           Qualifiers, Enumerated Types, Structures & Unions, Typedef
## #module

**Syntax:**  
```
#MODULE
```

**Elements:** None

**Purpose:** All global symbols created from the `#MODULE` to the end of the file will only be visible within that same block of code (and files `#INCLUDE` within that block). This may be used to limit the scope of global variables and functions within include files. This directive also applies to pre-processor #defines.

Note: The `extern` and `static` data qualifiers can also be used to denote scope of variables and functions as in the standard C methodology. `#MODULE` does add some benefits in that pre-processor #DEFINE can be given scope, which cannot normally be done in standard C methodology.

**Examples:**
```c
int GetCount(void);
void SetCount(int newCount);

#MODULE
int g_count;
#define G_COUNT_MAX 100
int GetCount(void) {return(g_count);}
void SetCount(int newCount) {
    if (newCount>G_COUNT_MAX)
        newCount=G_COUNT_MAX;
    g_count=newCount;
}
/*
the functions GetCount() and SetCount() have global scope, but the variable g_count and the #define G_COUNT_MAX only has scope to this file.
*/
```

**Example Files:** None

**See Also:**  
`#EXPORT`,  
[Invoking the Command Line Compiler](#),  
[Multiple Compilation Unit](#)
### #nolist

**Syntax:**

```
#NOLIST
```

**Elements:** None

**Purpose:** Stops inserting source lines into the .LST file (until a #LIST)

**Examples:**

```
#NOLIST  // Don't clutter up the list file
#include <cdriver.h>
#LIST
```

**Example Files:** 16c74.h

**Also See:** #LIST

### #ocs

**Syntax:**

```
#OCS x
```

**Elements:**

`x` is the clock's speed and can be 1 Hz to 100 MHz.

**Purpose:** Used instead of the `#use delay(clock = x)`

**Examples:**

```
#include <18F4520.h>
#device ICD=TRUE
#OCS 20 MHz
#use rs232(debugger)

void main() {
    ------;
}
```

**Example Files:** None

**Also See:** #USE DELAY
<table>
<thead>
<tr>
<th><strong>#opt</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax:</strong></td>
</tr>
<tr>
<td><strong>Elements:</strong></td>
</tr>
<tr>
<td><strong>Purpose:</strong></td>
</tr>
<tr>
<td><strong>Examples:</strong></td>
</tr>
<tr>
<td><strong>Example Files:</strong></td>
</tr>
<tr>
<td><strong>Also See:</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>#org</strong></th>
</tr>
</thead>
</table>
| **Syntax:** | #ORG start, end  
or  
#ORG segment  
or  
#ORG start, end { }  
or  
#ORG start, end auto=0  
#ORG start,end DEFAULT  
or  
#ORG DEFAULT |
| **Elements:** | start is the first ROM location (word address) to use, end is the last ROM location, segment is the start ROM location from a previous #ORG |
| **Purpose:** | This directive will fix the following function, constant or ROM declaration into a specific ROM area. End may be omitted if a segment was previously defined if you only want to add another function to the segment.  
Follow the ORG with a { } to only reserve the area with nothing inserted by the compiler. |
The RAM for a ORG’d function may be reset to low memory so the local variables and scratch variables are placed in low memory. This should only be used if the ORG’d function will not return to the caller. The RAM used will overlap the RAM of the main program. Add a AUTO=0 at the end of the #ORG line.

If the keyword DEFAULT is used then this address range is used for all functions user and compiler generated from this point in the file until a #ORG DEFAULT is encountered (no address range). If a compiler function is called from the generated code while DEFAULT is in effect the compiler generates a new version of the function within the specified address range.

#ORG may be used to locate data in ROM. Because CONSTANT are implemented as functions the #ORG should proceed the CONSTANT and needs a start and end address. For a ROM declaration only the start address should be specified.

When linking multiple compilation units be aware this directive applies to the final object file. It is an error if any #ORG overlaps between files unless the #ORG matches exactly.

**Examples:**

```
#ORG 0x1E00, 0x1FFF
MyFunc() {
  //This function located at 1E00
}

#ORG 0x1E00
AnotherFunc(){
  // This will be somewhere 1E00-1F00
}

#ORG 0x800, 0x820 {}  //Nothing will be at 800-820

#ORG 0x1B80
ROM int32 seridl_N0=12345;

#ORG 0x1C00, 0x1C0F
CHAR CONST ID[10]= {"123456789");
  //This ID will be at 1C00
  //Note some extra code will proceed the 123456789

#ORG 0x1F00, 0x1FF0
Void loader (){.
```

Example Files:  loader.c

Also See:  #ROM

#pin_select

Syntax:  #PIN_SELECT function=pin_xx

Elements:  function is the Microchip defined pin function name, such as: U1RX (UART1 receive), INT1 (external interrupt 1), T2CK (timer 2 clock), IC1 (input capture 1), OC1 (output capture 1).

<table>
<thead>
<tr>
<th>NULL</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1OUT</td>
<td>Comparator 1 Output</td>
</tr>
<tr>
<td>C2OUT</td>
<td>Comparator 2 Output</td>
</tr>
<tr>
<td>C3OUT</td>
<td>Comparator 3 Output</td>
</tr>
<tr>
<td>C4OUT</td>
<td>Comparator 4 Output</td>
</tr>
<tr>
<td>U1TX</td>
<td>UART1 Transmit</td>
</tr>
<tr>
<td>U1RTS</td>
<td>UART1 Request to Send</td>
</tr>
<tr>
<td>U2TX</td>
<td>UART2 Transmit</td>
</tr>
<tr>
<td>U2RTS</td>
<td>UART2 Request to Send</td>
</tr>
<tr>
<td>U3TX</td>
<td>UART3 Transmit</td>
</tr>
<tr>
<td>U3RTS</td>
<td>UART3 Request to Send</td>
</tr>
<tr>
<td>U4TX</td>
<td>UART4 Transmit</td>
</tr>
<tr>
<td>U4RTS</td>
<td>UART4 Request to Send</td>
</tr>
<tr>
<td>SDO1</td>
<td>SPI1 Data Output</td>
</tr>
<tr>
<td>SCK1OUT</td>
<td>SPI1 Clock Output</td>
</tr>
<tr>
<td>SS1OUT</td>
<td>SPI1 Slave Select Output</td>
</tr>
<tr>
<td>SDO2</td>
<td>SPI2 Data Output</td>
</tr>
<tr>
<td>SCK2OUT</td>
<td>SPI2 Clock Output</td>
</tr>
<tr>
<td>SS2OUT</td>
<td>SPI2 Slave Select Output</td>
</tr>
<tr>
<td>SDO3</td>
<td>SPI3 Data Output</td>
</tr>
<tr>
<td>SCK3OUT</td>
<td>SPI3 Clock Output</td>
</tr>
<tr>
<td>SS3OUT</td>
<td>SPI3 Slave Select Output</td>
</tr>
<tr>
<td>SDO4</td>
<td>SPI4 Data Output</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>SCK4OUT</td>
<td>SPI4 Clock Output</td>
</tr>
<tr>
<td>SS4OUT</td>
<td>SPI4 Slave Select Output</td>
</tr>
<tr>
<td>OC1</td>
<td>Output Compare 1</td>
</tr>
<tr>
<td>OC2</td>
<td>Output Compare 2</td>
</tr>
<tr>
<td>OC3</td>
<td>Output Compare 3</td>
</tr>
<tr>
<td>OC4</td>
<td>Output Compare 4</td>
</tr>
<tr>
<td>OC5</td>
<td>Output Compare 5</td>
</tr>
<tr>
<td>OC6</td>
<td>Output Compare 6</td>
</tr>
<tr>
<td>OC7</td>
<td>Output Compare 7</td>
</tr>
<tr>
<td>OC8</td>
<td>Output Compare 8</td>
</tr>
<tr>
<td>OC9</td>
<td>Output Compare 9</td>
</tr>
<tr>
<td>OC10</td>
<td>Output Compare 10</td>
</tr>
<tr>
<td>OC11</td>
<td>Output Compare 11</td>
</tr>
<tr>
<td>OC12</td>
<td>Output Compare 12</td>
</tr>
<tr>
<td>OC13</td>
<td>Output Compare 13</td>
</tr>
<tr>
<td>OC14</td>
<td>Output Compare 14</td>
</tr>
<tr>
<td>OC15</td>
<td>Output Compare 15</td>
</tr>
<tr>
<td>OC16</td>
<td>Output Compare 16</td>
</tr>
<tr>
<td>C1TX</td>
<td>CAN1 Transmit</td>
</tr>
<tr>
<td>C2TX</td>
<td>CAN2 Transmit</td>
</tr>
<tr>
<td>CSDO</td>
<td>DCI Serial Data Output</td>
</tr>
<tr>
<td>CSCKOUT</td>
<td>DCI Serial Clock Output</td>
</tr>
<tr>
<td>COFSOUT</td>
<td>DCI Frame Sync Output</td>
</tr>
<tr>
<td>UPDN1</td>
<td>QEI1 Direction Status Output</td>
</tr>
<tr>
<td>UPDN2</td>
<td>QEI2 Direction Status Output</td>
</tr>
<tr>
<td>CTPLS</td>
<td>CTMU Output Pulse</td>
</tr>
<tr>
<td>SYNCO1</td>
<td>PWM Synchronization Output Signal</td>
</tr>
<tr>
<td>SYNCO2</td>
<td>PWM Secondary Synchronization Output Signal</td>
</tr>
<tr>
<td>REFCLKO</td>
<td>REFCLK Output Signal</td>
</tr>
<tr>
<td>CMP1</td>
<td>Analog Comparator Output 1</td>
</tr>
<tr>
<td>CMP2</td>
<td>Analog Comparator Output 2</td>
</tr>
<tr>
<td>CMP3</td>
<td>Analog Comparator Output 3</td>
</tr>
<tr>
<td>CMP4</td>
<td>Analog Comparator Output 4</td>
</tr>
<tr>
<td>PWM4H</td>
<td>PWM4 High Output</td>
</tr>
<tr>
<td>PWM4L</td>
<td>PWM4 Low Output</td>
</tr>
<tr>
<td>QEI1CCMP</td>
<td>QEI1 Counter Comparator Output</td>
</tr>
<tr>
<td>QEI2CCMP</td>
<td>QEI2 Counter Comparator Output</td>
</tr>
<tr>
<td>MDOUT</td>
<td>DSM Modulator Output</td>
</tr>
<tr>
<td>DCIDO</td>
<td>DCI Serial Data Output</td>
</tr>
<tr>
<td>DCISCKOUT</td>
<td>DCI Serial Clock Output</td>
</tr>
<tr>
<td>DCIFSOUT</td>
<td>DCI Frame Sync Output</td>
</tr>
<tr>
<td>INT1</td>
<td>External Interrupt 1 Input</td>
</tr>
<tr>
<td>INT2</td>
<td>External Interrupt 2 Input</td>
</tr>
<tr>
<td>PIN</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>INT3</td>
<td>External Interrupt 3 Input</td>
</tr>
<tr>
<td>INT4</td>
<td>External Interrupt 4 Input</td>
</tr>
<tr>
<td>T1CK</td>
<td>Timer 1 External Clock Input</td>
</tr>
<tr>
<td>T2CK</td>
<td>Timer 2 External Clock Input</td>
</tr>
<tr>
<td>T3CK</td>
<td>Timer 3 External Clock Input</td>
</tr>
<tr>
<td>T4CK</td>
<td>Timer 4 External Clock Input</td>
</tr>
<tr>
<td>T5CK</td>
<td>Timer 5 External Clock Input</td>
</tr>
<tr>
<td>T6CK</td>
<td>Timer 6 External Clock Input</td>
</tr>
<tr>
<td>T7CK</td>
<td>Timer 7 External Clock Input</td>
</tr>
<tr>
<td>T8CK</td>
<td>Timer 8 External Clock Input</td>
</tr>
<tr>
<td>T9CK</td>
<td>Timer 9 External Clock Input</td>
</tr>
<tr>
<td>IC1</td>
<td>Input Capture 1</td>
</tr>
<tr>
<td>IC2</td>
<td>Input Capture 2</td>
</tr>
<tr>
<td>IC3</td>
<td>Input Capture 3</td>
</tr>
<tr>
<td>IC4</td>
<td>Input Capture 4</td>
</tr>
<tr>
<td>IC5</td>
<td>Input Capture 5</td>
</tr>
<tr>
<td>IC6</td>
<td>Input Capture 6</td>
</tr>
<tr>
<td>IC7</td>
<td>Input Capture 7</td>
</tr>
<tr>
<td>IC8</td>
<td>Input Capture 8</td>
</tr>
<tr>
<td>IC9</td>
<td>Input Capture 9</td>
</tr>
<tr>
<td>IC10</td>
<td>Input Capture 10</td>
</tr>
<tr>
<td>IC11</td>
<td>Input Capture 11</td>
</tr>
<tr>
<td>IC12</td>
<td>Input Capture 12</td>
</tr>
<tr>
<td>IC13</td>
<td>Input Capture 13</td>
</tr>
<tr>
<td>IC14</td>
<td>Input Capture 14</td>
</tr>
<tr>
<td>IC15</td>
<td>Input Capture 15</td>
</tr>
<tr>
<td>IC16</td>
<td>Input Capture 16</td>
</tr>
<tr>
<td>C1RX</td>
<td>CAN1 Receive</td>
</tr>
<tr>
<td>C2RX</td>
<td>CAN2 Receive</td>
</tr>
<tr>
<td>OCFA</td>
<td>Output Compare Fault A Input</td>
</tr>
<tr>
<td>OCFB</td>
<td>Output Compare Fault B Input</td>
</tr>
<tr>
<td>OCFC</td>
<td>Output Compare Fault C Input</td>
</tr>
<tr>
<td>U1RX</td>
<td>UART1 Receive</td>
</tr>
<tr>
<td>U1CTS</td>
<td>UART1 Clear to Send</td>
</tr>
<tr>
<td>U2RX</td>
<td>UART2 Receive</td>
</tr>
<tr>
<td>U2CTS</td>
<td>UART2 Clear to Send</td>
</tr>
<tr>
<td>U3RX</td>
<td>UART3 Receive</td>
</tr>
<tr>
<td>U3CTS</td>
<td>UART3 Clear to Send</td>
</tr>
<tr>
<td>U4RX</td>
<td>UART4 Receive</td>
</tr>
<tr>
<td>U4CTS</td>
<td>UART4 Clear to Send</td>
</tr>
<tr>
<td>SDI1</td>
<td>SPI1 Data Input</td>
</tr>
<tr>
<td>SCK1IN</td>
<td>SPI1 Clock Input</td>
</tr>
<tr>
<td>SS1IN</td>
<td>SPI1 Slave Select Input</td>
</tr>
<tr>
<td>SDI2</td>
<td>SPI2 Data Input</td>
</tr>
<tr>
<td>Pin</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>SCK2IN</td>
<td>SPI2 Clock Input</td>
</tr>
<tr>
<td>SS2IN</td>
<td>SPI2 Slave Select Input</td>
</tr>
<tr>
<td>SDI3</td>
<td>SPI3 Data Input</td>
</tr>
<tr>
<td>SCK3IN</td>
<td>SPI3 Clock Input</td>
</tr>
<tr>
<td>SS3IN</td>
<td>SPI3 Slave Select Input</td>
</tr>
<tr>
<td>SDI4</td>
<td>SPI4 Data Input</td>
</tr>
<tr>
<td>SCK4IN</td>
<td>SPI4 Clock Input</td>
</tr>
<tr>
<td>SS4IN</td>
<td>SPI4 Slave Select Input</td>
</tr>
<tr>
<td>CSDI</td>
<td>DCI Serial Data Input</td>
</tr>
<tr>
<td>CSCK</td>
<td>DCI Serial Clock Input</td>
</tr>
<tr>
<td>COFS</td>
<td>DCI Frame Sync Input</td>
</tr>
<tr>
<td>FLTA1</td>
<td>PWM1 Fault Input</td>
</tr>
<tr>
<td>FLTA2</td>
<td>PWM2 Fault Input</td>
</tr>
<tr>
<td>QEA1</td>
<td>QEi1 Phase A Input</td>
</tr>
<tr>
<td>QEA2</td>
<td>QEi2 Phase A Input</td>
</tr>
<tr>
<td>QEB1</td>
<td>QEi1 Phase B Input</td>
</tr>
<tr>
<td>QEB2</td>
<td>QEi2 Phase B Input</td>
</tr>
<tr>
<td>INDEX1</td>
<td>QEi1 Index Input</td>
</tr>
<tr>
<td>INDEX2</td>
<td>QEi2 Index Input</td>
</tr>
<tr>
<td>HOME1</td>
<td>QEi1 Home Input</td>
</tr>
<tr>
<td>HOME2</td>
<td>QEi2 Home Input</td>
</tr>
<tr>
<td>FLT1</td>
<td>PWM1 Fault Input</td>
</tr>
<tr>
<td>FLT2</td>
<td>PWM2 Fault Input</td>
</tr>
<tr>
<td>FLT3</td>
<td>PWM3 Fault Input</td>
</tr>
<tr>
<td>FLT4</td>
<td>PWM4 Fault Input</td>
</tr>
<tr>
<td>FLT5</td>
<td>PWM5 Fault Input</td>
</tr>
<tr>
<td>FLT6</td>
<td>PWM6 Fault Input</td>
</tr>
<tr>
<td>FLT7</td>
<td>PWM7 Fault Input</td>
</tr>
<tr>
<td>FLT8</td>
<td>PWM8 Fault Input</td>
</tr>
<tr>
<td>SYNCI1</td>
<td>PWM Synchronization Input 1</td>
</tr>
<tr>
<td>SYNCI2</td>
<td>PWM Synchronization Input 2</td>
</tr>
<tr>
<td>DCIDI</td>
<td>DCI Serial Data Input</td>
</tr>
<tr>
<td>DCISKIN</td>
<td>DCI Serial Clock Input</td>
</tr>
<tr>
<td>DCIFSIN</td>
<td>DCI Frame Sync Input</td>
</tr>
<tr>
<td>DTCMP1</td>
<td>PWM Dead Time Compensation 1 Input</td>
</tr>
<tr>
<td>DTCMP2</td>
<td>PWM Dead Time Compensation 2 Input</td>
</tr>
<tr>
<td>DTCMP3</td>
<td>PWM Dead Time Compensation 3 Input</td>
</tr>
<tr>
<td>DTCMP4</td>
<td>PWM Dead Time Compensation 4 Input</td>
</tr>
<tr>
<td>DTCMP5</td>
<td>PWM Dead Time Compensation 5 Input</td>
</tr>
<tr>
<td>DTCMP6</td>
<td>PWM Dead Time Compensation 6 Input</td>
</tr>
<tr>
<td>DTCMP7</td>
<td>PWM Dead Time Compensation 7 Input</td>
</tr>
</tbody>
</table>

*pin_xx* is the CCS provided pin definition. For example: PIN_C7,
PIN_B0, PIN_D3, etc.

**Purpose:** On PICs that contain Peripheral Pin Select (PPS), this allows the programmer to define which pin a peripheral is mapped to.

**Examples:**
```
#pin_select U1TX=PIN_C6
#pin_select U1RX=PIN_C7
#pin_select INT1=PIN_B0
```

**Example Files:** None

**Also See:** pin_select()

---

__pcd__

**Syntax:**
```
__PCD__
```

**Elements:** None

**Purpose:** The PCD compiler defines this pre-processor identifier. It may be used to determine if the PCD compiler is doing the compilation.

**Examples:**
```
#ifdef __pcd__
#device dsPIC33FJ256MC710
#endif
```

**Example Files:** ex_sqw.c

**Also See:** None

---

**pragma**

**Syntax:**
```
#PRAGMA cmd
```

**Elements:** `cmd` is any valid preprocessor directive.

**Purpose:** This directive is used to maintain compatibility between C compilers. This compiler will accept this directive before any other preprocessor command. In no case does this compiler require this directive.
#profile

Syntax:  

```
#profile options
```

Elements:  

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>functions</strong></td>
<td>Profiles the start/end of functions and all <code>profileout()</code> messages.</td>
</tr>
<tr>
<td><strong>functions, parameters</strong></td>
<td>Profiles the start/end of functions, parameters sent to functions, and all <code>profileout()</code> messages.</td>
</tr>
<tr>
<td><code>profileout</code></td>
<td>Only <code>profileout()</code> messages.</td>
</tr>
<tr>
<td><code>paths</code></td>
<td>Profiles every branch in the code.</td>
</tr>
<tr>
<td><code>off</code></td>
<td>Disable all code profiling.</td>
</tr>
<tr>
<td><code>on</code></td>
<td>Re-enables the code profiling that was previously disabled with a <code>#profile off</code> command. This will use the last options before disabled with the off command.</td>
</tr>
</tbody>
</table>

Purpose:  

Large programs on the microcontroller may generate lots of profile data, which may make it difficult to debug or follow. By using `#profile` the user can dynamically control which points of the program are being profiled, and limit data to what is relevant to the user.

Examples:  

```
#profile off
void BigFunction(void)
{
    // BigFunction code goes here.
    // Since #profile off was called above,
    // no profiling will happen even for other
    // functions called by BigFunction().
}
#profile on
```
#recursive

Syntax:  

```
#RECURSIVE
```

Elements: None

Purpose: Tells the compiler that the procedure immediately following the directive will be recursive.

Examples:

```
int factorial(int num) {
  if (num <= 1)
    return 1;
  return num * factorial(num-1);
}
```

Example Files: None

Also See: None

#reserve

Syntax:  

```
#RESERVE address
or
#RESERVE address, address, address
or
#RESERVE start:end
```

Elements: `address` is a RAM address, `start` is the first address and `end` is the last address.

Purpose: This directive allows RAM locations to be reserved from use by the compiler. `#RESERVE` must appear after the `#DEVICE` otherwise it will have no effect. When linking multiple compilation units be aware this
directive applies to the final object file.

**Examples:**

```
#DEVICE dsPIC30F2010
#RESERVE 0x800:0xB3
```

**Example Files:**

`ex_cust.c`

**Also See:**

`#ORG`

---

## #rom

**Syntax:**

```
#ROM address = {list}
#ROM type address = {list}
```

**Elements:**

- `address` is a ROM word address, `list` is a list of words separated by commas

**Purpose:**

Allows the insertion of data into the .HEX file. In particular, this may be used to program the '84 data EEPROM, as shown in the following example.

Note that if the #ROM address is inside the program memory space, the directive creates a segment for the data, resulting in an error if a #ORG is over the same area. The #ROM data will also be counted as used program memory space.

The type option indicates the type of each item, the default is 16 bits. Using char as the type treats each item as 7 bits packing 2 chars into every pcm 14-bit word.

When linking multiple compilation units be aware this directive applies to the final object file.

Some special forms of this directive may be used for verifying program memory:

```
#ROM address = checksum
    This will put a value at address such that the entire program memory will sum to 0x1248

#ROM address = crc16
```
This will put a value at address that is a crc16 of all the program memory except the specified address

#ROM address = crc16(start, end)
This will put a value at address that is a crc16 of all the program memory from start to end.

#ROM address = crc8
This will put a value at address that is a crc16 of all the program memory except the specified address

Examples:
#rom getnev ("EEPROM_ADDRESS")={1,2,3,4,5,6,7,8}
#rom int8 0x1000="(c)CCS, 2010"

Example Files: None

Also See: #ORG

#separate

Syntax: #SEPARATE options

Elements: options is optional, and are:

STDCALL – Use the standard Microchip calling method, used in C30. W0-W7 is used for function parameters, rest of the working registers are not touched, remaining function parameters are pushed onto the stack.

ARG=Wx:Wy – Use the working registers Wx to Wy to hold function parameters. Any remaining function parameters are pushed onto the stack.

DND=Wx:Wy – Function will not change Wx to Wy working registers.

AVOID=Wx:Wy – Function will not use Wx to Wy working registers for function parameters.

NO RETURN - Prevents the compiler generated return at the end of a function.

You cannot use STDCALL with the ARG, DND or AVOID parameters.
If you do not specify one of these options, the compiler will determine the best configuration, and will usually not use the stack for function parameters (usually scratch space is allocated for parameters).

**Purpose:**
Tells the compiler that the procedure IMMEDIATELY following the directive is to be implemented SEPARATELY. This is useful to prevent the compiler from automatically making a procedure INLINE. This will save ROM space but it does use more stack space. The compiler will make all procedures marked SEPARATE, separate, as requested, even if there is not enough stack space to execute.

**Examples:**
```c
#separate ARG=W0:W7 AVOID=W8:W15 DND=W8:W15
swapbyte (int *a, int *b) {
    int t;
    t=*a;
    *a=*b;
    *b=t;
}
```

**Example Files:** ex_cust.c

**Also See:** #INLINE

---

### #serialize

**Syntax:**
```
#SERIALIZE(id=xxx, next="x" | file="filename.txt" | listfile="filename.txt", "prompt="text", log="filename.txt") -
or
#SERIALIZE(dataee=x, binary=x, next="x" | file="filename.txt" | listfile="filename.txt", prompt="text", log="filename.txt")
```

**Elements:**
- **id=xxx** - Specify a C CONST identifier, may be int8, int16, int32 or char array
- **dataee=x** - The address x is the start address in the data EEPROM.
- **binary=x** - The integer x is the number of bytes to be written to address specified.
- **string=x** - The integer x is the number of bytes to be written to address specified.
**unicode=n** - If n is a 0, the string format is normal unicode. For n>0 n indicates the string number in a USB descriptor.

Use only one of the next three options:

**file=“filename.txt”** - The file x is used to read the initial serial number from, and this file is updated by the ICD programmer. It is assumed this is a one line file with the serial number. The programmer will increment the serial number.

**listfile=“filename.txt”** - The file x is used to read the initial serial number from, and this file is updated by the ICD programmer. It is assumed this is a file one serial number per line. The programmer will read the first line then delete that line from the file.

**next=“x”** - The serial number X is used for the first load, then the hex file is updated to increment x by one.

Other optional parameters:

**prompt=“text”** - If specified the user will be prompted for a serial number on each load. If used with one of the above three options then the default value the user may use is picked according to the above rules.

**log=xxx** - A file may optionally be specified to keep a log of the date, time, hex file name and serial number each time the part is programmed. If no id=xxx is specified then this may be used as a simple log of all loads of the hex file.

**Purpose:** Assists in making serial numbers easier to implement when working with CCS ICD units. Comments are inserted into the hex file that the ICD software interprets.

**Examples:**

```c
//Prompt user for serial number to be placed
//at address of serialNumA
//Default serial number = 200
int8 const serialNumA=100;
#serialize(id=serialNumA,next="200",prompt="Enter the serial number")

//Adds serial number log in seriallog.txt
#serialize(id=serialNumA,next="200",prompt="Enter the serial number", log="seriallog.txt")

//Retrieves serial number from serials.txt
#serialize(id=serialNumA,listfile="serials.txt")
```
//Place serial number at EEPROM address 0, reserving 1 byte
#serialize(dataee=0,binary=1,next="45",prompt="Put in Serial number")

//Place string serial number at EEPROM address 0, reserving 2 bytes
#serialize(dataee=0, string=2,next="AB",prompt="Put in Serial number")

Example Files: None

Also See: None

#task

(The RTOS is only included with the PCW, PCWH, and PCWHD software packages.)

Each RTOS task is specified as a function that has no parameters and no return. The #TASK directive is needed just before each RTOS task to enable the compiler to tell which functions are RTOS tasks. An RTOS task cannot be called directly like a regular function can.

Syntax:  #TASK (options)

Elements: options are separated by comma and may be:
rate=time
Where time is a number followed by s, ms, us, or ns. This specifies how often the task will execute.

max=time
Where time is a number followed by s, ms, us, or ns. This specifies the budgeted time for this task.

queue=bytes
Specifies how many bytes to allocate for this task’s incoming messages. The default value is 0.

enabled=value
Specifies whether a task is enabled or disabled by rtos_run( ). True for enabled, false for disabled. The default value is enabled.

Purpose: This directive tells the compiler that the following function is an RTOS
task.

The rate option is used to specify how often the task should execute. This must be a multiple of the minor cycle option if one is specified in the #USE RTOS directive.

The max option is used to specify how much processor time a task will use in one execution of the task. The time specified in max must be equal to or less than the time specified in the minor cycle option of the #USE RTOS directive before the project will compile successfully. The compiler does not have a way to enforce this limit on processor time, so a programmer must be careful with how much processor time a task uses for execution. This option does not need to be specified.

The queue option is used to specify the number of bytes to be reserved for the task to receive messages from other tasks or functions. The default queue value is 0.

Examples:  
\[\text{#task(rate=1s, max=20ms, queue=5)}\]

Also See:  
\[\text{#USE RTOS}\]

___time___

Syntax:  
\[\text{__TIME__}\]

Elements: None

Purpose: This pre-processor identifier is replaced at compile time with the time of the compile in the form: "hh:mm:ss"

Examples:  
\[\text{printf("Software was compiled on ");}\]
\[\text{printf(__TIME__);}\]

Example Files: None

Also See: None
#type

## Syntax:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard-type=size</td>
<td>specify the type and size</td>
</tr>
<tr>
<td>default=area</td>
<td>specify the default area</td>
</tr>
<tr>
<td>unsigned</td>
<td>set the type to unsigned</td>
</tr>
<tr>
<td>signed</td>
<td>set the type to signed</td>
</tr>
<tr>
<td>char=sed</td>
<td>set the type of char to signed</td>
</tr>
<tr>
<td>char=unsigned</td>
<td>set the type of char to unsigned</td>
</tr>
<tr>
<td>ARG=Wx:Wy</td>
<td>specify the range of working registers</td>
</tr>
<tr>
<td>DND=Wx:Wy</td>
<td>specify the default area</td>
</tr>
<tr>
<td>AVOID=Wx:Wy</td>
<td>specify the area to avoid</td>
</tr>
<tr>
<td>RECURSIVE</td>
<td>set the type to recursive</td>
</tr>
<tr>
<td>CLASSIC</td>
<td>set the type to classic</td>
</tr>
</tbody>
</table>

## Elements:
- **standard-type** is one of the C keywords short, int, long, float, or double
- **size** is 1, 8, 16, 48, or 64
- **area** is a memory region defined before the #TYPE using the addressmod directive

Wx:Wy is a range of working registers (example: W0, W1, W15, etc)

## Purpose:

By default the compiler treats SHORT as 8 bits, INT as 16 bits, and LONG as 32 bits. The traditional C convention is to have INT defined as the most efficient size for the target processor. This is why it is 16 bits on the dsPIC/PIC24®. In order to help with code compatibility a #TYPE directive may be used to allow these types to be changed. #TYPE can redefine these keywords.

Note that the commas are optional. Be warned CCS example programs and include files may not work right if you use #TYPE in your program.

Classic will set the type sizes to be compatible with CCS PIC programs.

This directive may also be used to change the default RAM area used for variable storage. This is done by specifying default=area where area is a addressmod address space.

When linking multiple compilation units be aware this directive only applies to the current compilation unit.

The #TYPE directive allows the keywords UNSIGNED and SIGNED to set the default data type.
The ARG parameter tells the compiler that all functions can use those working registers to receive parameters. The DND parameters tells the compiler that all functions should not change those working registers (not use them for scratch space). The AVOID parameter tells the compiler to not use those working registers for passing variables to functions. If you are using recursive functions, then it will use the stack for passing variables when there is not enough working registers to hold variables; if you are not using recursive functions, the compiler will allocate scratch space for holding variables if there is not enough working registers. #SEPARATE can be used to set these parameters on an individual basis.

The RECURSIVE option tells the compiler that ALL functions can be recursive. #RECURSIVE can also be used to assign this status on an individual basis.

### Examples:

```
#TYPE  SHORT=  1 , INT=  8 , LONG= 16, FLOAT=48

#TYPE default=area

addressmod (user_ram_block, 0x100, 0x1FF);

#type default=user_ram_block // all variable declarations
    // in this area will be in
    // 0x100-0x1FF

#type default= // restores memory allocation
    // back to normal

#TYPE SIGNED

#TYPE RECURSIVE

#TYPE ARG=W0:W7
#TYPE AVOID=W8:W15
#TYPE DND=W8:W15

... void main()
{
    int variable1; // variable1 can only take values from -128 to 127
    ...
    ...
}
```

Example Files: ex_cust.c

Also See: None
#undef

Syntax:  
#UNDEF id

Elements:  id is a pre-processor id defined via #DEFINE

Purpose:  The specified pre-processor ID will no longer have meaning to the pre-processor.

Examples:  
#if MAXSIZE<100
 #undef MAXSIZE
 #define MAXSIZE 100
#endif

Example Files:  None

Also See:  #DEFINE

__unicode

Syntax:  
__unicode( constant-string )

Elements:  Unicode format string

Purpose  
This macro will convert a standard ASCII string to a Unicode format string by inserting a \000 after each character and removing the normal C string terminator.

For example:  __unicode("ABCD")
will return:  "A\00B\000C\000D" (8 bytes total with the terminator)

Since the normal C terminator is not used for these strings you need to do one of the following for variable length strings:
string = _unicode(KEYWORD) "\000\000";
OR
string = _unicode(KEYWORD);
string_size = sizeof(_unicode(KEYWORD));

Examples:
#define USB_DESC_STRING_TYPE 3
#define USB_STRING(x)
(sizeof(_unicode(x))+2),USB_DESC_STRING_TYPE,_unicode(x)
#define USB_ENGLISH_STRING
4,USB_DESC_STRING_TYPE,0x09,0x04
  //Microsoft

Defined for US-English

char const USB_STRING_DESC[]={[USB_ENGLISH_STRING,
USB_STRING("CCS"),
USB_STRING("CCS HID DEMO")};

Example
Files: usb_desc_hid.h

#use capture

Syntax: #USE CAPTURE(options)

Elements: ICx/CCPx
Which CCP/Input Capture module to us.

INPUT = PIN_xx
Specifies which pin to use. Useful for device with
remappable pins, this will cause compiler to automatically
assign pin to peripheral.

TIMER=x
Specifies the timer to use with capture unit. If not specified
default to timer 1 for PCM and PCH compilers and timer 3
for PCD compiler.

TICK=x
The tick time to setup the timer to. If not specified it will be
set to fastest as possible or if same timer was already setup by a previous stream it will be set to that tick time. If using same timer as previous stream and different tick time an error will be generated.

**FASTEST**
Use instead of TICK=x to set tick time to fastest as possible.

**SLOWEST**
Use instead of TICK=x to set tick time to slowest as possible.

**CAPTURE_RISING**
Specifies the edge that timer value is captured on. Defaults to CAPTURE_RISING.

**CAPTURE_FALLING**
Specifies the edge that timer value is captured on. Defaults to CAPTURE_RISING.

**CAPTURE_BOTH**
PCD only. Specifies the edge that timer value is captured on. Defaults to CAPTURE_RISING.

**PRE=x**
Specifies number of rising edges before capture event occurs. Valid options are 1, 4 and 16, default to 1 if not specified. Options 4 and 16 are only valid when using CAPTURE_RISING, will generate an error if used with CAPTURE_FALLING or CAPTURE_BOTH.

**ISR=x**
PCD only. Specifies the number of capture events to occur before generating capture interrupt. Valid options are 1, 2, 3 and 4, defaults to 1 is not specified. Option 1 is only valid option when using CAPTURE_BOTH, will generate an error if trying to use 2, 3 or 4 with it.

**STREAM=id**
Associates a stream identifier with the capture module. The identifier may be used in functions like get_capture_time().

**DEFINE=id**
<table>
<thead>
<tr>
<th>PreProcessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creates a define named id which specifies the number of capture per second. Default define name if not specified is CAPTURES_PER_SECOND. Define name must start with an ASCII letter 'A' to 'Z', an ASCII letter 'a' to 'z' or an ASCII underscore ('_').</td>
</tr>
<tr>
<td><strong>Purpose:</strong></td>
</tr>
<tr>
<td>This directive tells the compiler to setup an input capture on the specified pin using the specified settings. The #USE DELAY directive must appear before this directive can be used. This directive enables use of built-in functions such as get_capture_time() and get_capture_event().</td>
</tr>
<tr>
<td><strong>Examples:</strong></td>
</tr>
<tr>
<td>#USE CAPTURE(INPUT=PIN_C2,CAPTURE_RISING,TIMER=1, FASTEST)</td>
</tr>
<tr>
<td><strong>Example Files:</strong></td>
</tr>
<tr>
<td>None.</td>
</tr>
<tr>
<td><strong>Also See:</strong></td>
</tr>
<tr>
<td>get_capture_time(), get_capture_event()</td>
</tr>
</tbody>
</table>

**#use delay**

**Syntax:**

#USE DELAY (options))

**Elements:** Options may be any of the following separated by commas:

- **clock=speed** speed is a constant 1-100000000 (1 hz to 100 mhz). This number can contains commas. This number also supports the following denominations: M, MHZ, K, KHZ. This specifies the clock the CPU runs at. Depending on the PIC this is 2 or 4 times the instruction rate. This directive is not needed if the following type=speed is used and there is no frequency multiplication or division.

- **type=speed** type defines what kind of clock you are using, and the following values are valid: oscillator, osc (same as oscillator), crystal, xtal (same as crystal), internal, int (same as internal) or rc. The compiler will automatically set the oscillator configuration bits based upon your defined type. If you specified internal, the compiler will also automatically set the internal oscillator to the defined speed. Configuration fuses are modified when this option is used. Speed is the input frequency.

- **restart_wdt** will restart the watchdog timer on every delay_us() and delay_ms() use.
**ACT or ACT=type** for device with Active Clock Tuning, type can be either USB or SOSC. If only using ACT type will default to USB. ACT=USB causes the compiler to enable the active clock tuning and to tune the internal oscillator to the USB clock. ACT=SOSC causes the compiler to enable the active clock tuning and to tune the internal oscillator to the secondary clock at 32.768 kHz. ACT can only be used when the system clock is set to run from the internal oscillator.

**AUX: type=speed** Some chips have a second oscillator used by specific peripherals and when this is the case this option sets up that oscillator.

**PLL_WAIT** when used with a PLL clock, it causes the compiler to poll PLL ready flag and to only continue program execution when flag indicates that the PLL is ready.

Also See:  
delay_ms(), delay_us()
#use fast_io

Syntax: #USE FAST_IO (\textit{port})

Elements: \textit{port} is A, B, C, D, E, F, G, H, J or ALL

Purpose: Affects how the compiler will generate code for input and output instructions that follow. This directive takes effect until another \#use xxxx_IO directive is encountered. The fast method of doing I/O will cause the compiler to perform I/O without programming of the direction register. The compiler's default operation is the opposite of this command, the direction I/O will be set/cleared on each I/O operation. The user must ensure the direction register is set correctly via set_tris_X(). When linking multiple compilation units be aware this directive only applies to the current compilation unit.

Examples: \#use fast_io(A)

Example Files: ex_cust.c

Also See: \#USE FIXED_IO, \#USE STANDARD_IO, set_tris_X(), General Purpose I/O

#use fixed_io

Syntax: #USE FIXED_IO (\textit{port_outputs}=\textit{pin}, \textit{pin}?)

Elements: \textit{port} is A-G, \textit{pin} is one of the pin constants defined in the devices .h file.

Purpose: This directive affects how the compiler will generate code for input and output instructions that follow. This directive takes effect until another \#USE XXX_IO directive is encountered. The fixed method of doing I/O will cause the compiler to generate code to make an I/O pin either input or output every time it is used. The pins are programmed according to the information in this directive (not the operations actually performed). This saves a byte of RAM used in standard I/O. When linking multiple compilation units be aware this directive only applies to the current compilation unit.
### #use i2c

**Syntax:**

```
#USE I2C (options)
```

**Elements:**

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASTER</td>
<td>Sets to the master mode</td>
</tr>
<tr>
<td>MULTI_MASTER</td>
<td>Set the multi_master mode</td>
</tr>
<tr>
<td>SLAVE</td>
<td>Set the slave mode</td>
</tr>
<tr>
<td>SCL=pin</td>
<td>Specifies the SCL pin (pin is a bit address)</td>
</tr>
<tr>
<td>SDA=pin</td>
<td>Specifies the SDA pin</td>
</tr>
<tr>
<td>ADDRESS=nn</td>
<td>Specifies the slave mode address</td>
</tr>
<tr>
<td>FAST</td>
<td>Use the fast I2C specification.</td>
</tr>
<tr>
<td>FAST=nnnnnn</td>
<td>Sets the speed to nnnnnn hz</td>
</tr>
<tr>
<td>SLOW</td>
<td>Use the slow I2C specification.</td>
</tr>
<tr>
<td>RESTART_WDT</td>
<td>Restart the WDT while waiting in I2C_READ</td>
</tr>
<tr>
<td>FORCE_HW</td>
<td>Use hardware I2C functions.</td>
</tr>
<tr>
<td>FORCE_SW</td>
<td>Use software I2C functions.</td>
</tr>
<tr>
<td>NOFLOAT_HIGH</td>
<td>Does not allow signals to float high, signals are driven from low to high</td>
</tr>
<tr>
<td>SMBUS</td>
<td>Bus used is not I2C bus, but very similar</td>
</tr>
<tr>
<td>STREAM=id</td>
<td>Associates a stream identifier with this I2C port. The identifier may then be used in functions like i2c_read or i2c_write.</td>
</tr>
<tr>
<td>NO_STRETCH</td>
<td>Do not allow clock stretching</td>
</tr>
<tr>
<td>MASK=nn</td>
<td>Set an address mask for parts that support it</td>
</tr>
</tbody>
</table>
### I2C1
Instead of SCL= and SDA= this sets the pins to the first module

### I2C2
Instead of SCL= and SDA= this sets the pins to the second module

### NOINIT
No initialization of the I2C peripheral is performed. Use I2C_INIT() to initialize peripheral at run time.

Only some chips allow the following:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_HOLD</td>
<td>No ACK is sent until I2C_READ is called for data bytes (slave only)</td>
</tr>
<tr>
<td>ADDRESS_HOLD</td>
<td>No ACK is sent until I2C_read is called for the address byte (slave only)</td>
</tr>
<tr>
<td>SDA_HOLD</td>
<td>Min of 300ns holdtime on SDA a from SCL goes low</td>
</tr>
</tbody>
</table>

**Purpose:**
CCS offers support for the hardware-based I2C™ and a software-based master I2C™ device. (For more information on the hardware-based I2C module, please consult the datasheet for your target device; not all PICs support I2C™.

The I2C library contains functions to implement an I2C bus. The #USE I2C remains in effect for the I2C_START, I2C_STOP, I2C_READ, I2C_WRITE and I2C_POLL functions until another USE I2C is encountered. Software functions are generated unless the FORCE_HW is specified. The SLAVE mode should only be used with the built-in SSP. The functions created with this directive are exported when using multiple compilation units. To access the correct function use the stream identifier.

**Examples:**

```c
#use I2C(master, sda=PIN_B0, scl=PIN_B1)

#use I2C(slave, sda=PIN_C4, scl=PIN_C3
  address=0xa0, FORCE_HW)

#use I2C(master, scl=PIN_B0, sda=PIN_B1, fast=450000)
  //sets the target speed to 450 KBSP
```

**Example Files:** [ex_extee.c](#) with [16c74.h](#)

**Also See:** [i2c_poll, i2c_speed, i2c_start, i2c_stop, i2c_slaveaddr, i2c_isr_state, i2c_write, i2c_read, I2C Overview](#)
#use profile()

**Syntax:**  
#use profile(options)

**Elements:**  
*options* may be any of the following, comma separated:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICD</td>
<td>Default – configures code profiler to use the ICD connection.</td>
</tr>
<tr>
<td>TIMER1</td>
<td>Optional. If specified, the code profiler run-time on the microcontroller will use the Timer1 peripheral as a timestamp for all profile events. If not specified the code profiler tool will use the PC clock, which may not be accurate for fast events.</td>
</tr>
<tr>
<td>BAUD=x</td>
<td>Optional. If specified, will use a different baud rate between the microcontroller and the code profiler tool. This may be required on slow microcontrollers to attempt to use a slower baud rate.</td>
</tr>
</tbody>
</table>

**Purpose:**  
Tell the compiler to add the code profiler run-time in the microcontroller and configure the link and clock.

**Examples:**  
#profile(ICD, TIMER1, baud=9600)

**Example Files:**  
ex_profile.c

**Also See:**  
#profile(), profileout(), Code Profile overview

---

#use pwm()

**Syntax:**  
#use pwm (options)

**Elements:**  
*options* are separated by commas and may be:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWMx or CCPx</td>
<td>Selects the CCP to use, x being the module number to use.</td>
</tr>
<tr>
<td>PWMx or OCx</td>
<td>Selects the Output Compare module, x being</td>
</tr>
</tbody>
</table>
the module number to use.

<table>
<thead>
<tr>
<th><strong>OUTPUT=PIN_xx</strong></th>
<th>Selects the PWM pin to use, pin must be one of the OC pins. If device has remappable pins compiler will assign specified pin to specified OC module. If OC module not specified it will assign remappable pin to first available module.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TIMER=x</strong></td>
<td>Selects timer to use with PWM module, default if not specified is timer 2.</td>
</tr>
<tr>
<td><strong>FREQUENCY=x</strong></td>
<td>Sets the period of PWM based off specified value, should not be used if PERIOD is already specified. If frequency can't be achieved exactly compiler will generate a message specifying the exact frequency and period of PWM. If neither FREQUENCY or PERIOD is specified, the period defaults to maximum possible period with maximum resolution and compiler will generate a message specifying the frequency and period of PWM, or if using same timer as previous stream instead of setting to maximum possible it will be set to the same as previous stream. If using same timer as previous stream and frequency is different compiler will generate an error.</td>
</tr>
<tr>
<td><strong>PERIOD=x</strong></td>
<td>Sets the period of PWM, should not be used if FREQUENCY is already specified. If period can't be achieved exactly compiler will generate a message specifying the exact period and frequency of PWM. If neither PERIOD or FREQUENCY is specified, the period defaults to maximum possible period with maximum resolution and compiler will generate a message specifying the frequency and period of PWM, or if using same timer as previous stream instead of setting to maximum possible it will be set to the same as previous stream. If using same timer as previous stream and period is different compiler will generate an error.</td>
</tr>
<tr>
<td><strong>BITS=x</strong></td>
<td>Sets the resolution of the the duty cycle, if period or frequency is specified will adjust the period to meet set resolution and will generate an message specifying the...</td>
</tr>
</tbody>
</table>
frequency and duty of PWM. If period or frequency not specified will set period to maximum possible for specified resolution and compiler will generate a message specifying the frequency and period of PWM, unless using same timer as previous then it will generate an error if resolution is different then previous stream. If not specified then frequency, period or previous stream using same timer sets the resolution.

**DUTY=x**
Selects the duty percentage of PWM, default if not specified is 50%.

**PWM_ON**
Initialize the PWM in the ON state, default state if pwm_on or pwm_off is not specified.

**PWM_OFF**
Initialize the PWM in the OFF state.

**STREAM=id**
Associates a stream identifier with the PWM signal. The identifier may be used in functions like pwm_set_duty_percent().

### Purpose:
This directive tells the compiler to setup a PWM on the specified pin using the specified frequency, period, duty cycle and resolution. The #USE DELAY directive must appear before this directive can be used. This directive enables use of built-in functions such as set_pwm_duty_percent(), set_pwm_frequency(), set_pwm_period(), pwm_on() and pwm_off().

### Examples:
None

### Also See:
pwm_on(), pwm_off(), pwm_set_frequency(), pwm_set_duty_percent(), pwm_set_duty()

---

### #use rs232

#### Syntax:
#USE RS232 *(options)*

#### Elements:
**Options** are separated by commas and may be:

<table>
<thead>
<tr>
<th>STREAM=id</th>
<th>Associates a stream identifier with this RS232 port. The identifier may then be used in functions like fputc.</th>
</tr>
</thead>
</table>

---

172
<table>
<thead>
<tr>
<th>PreProcessor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BAUD=x</strong></td>
<td>Set baud rate to x</td>
</tr>
<tr>
<td><strong>XMIT=pin</strong></td>
<td>Set transmit pin</td>
</tr>
<tr>
<td><strong>RCV=pin</strong></td>
<td>Set receive pin</td>
</tr>
<tr>
<td><strong>FORCE_SW</strong></td>
<td>Will generate software serial I/O routines even when the UART pins are specified.</td>
</tr>
<tr>
<td><strong>BRGH1OK</strong></td>
<td>Allow bad baud rates on chips that have baud rate problems.</td>
</tr>
<tr>
<td><strong>ENABLE=pin</strong></td>
<td>The specified pin will be high during transmit. This may be used to enable 485 transmit.</td>
</tr>
<tr>
<td><strong>DEBUGGER</strong></td>
<td>Indicates this stream is used to send/receive data through a CCS ICD unit. The default pin used is B3, use XMIT= and RCV= to change the pin used. Both should be the same pin.</td>
</tr>
<tr>
<td><strong>RESTART_WDT</strong></td>
<td>Will cause GETC() to clear the WDT as it waits for a character.</td>
</tr>
<tr>
<td><strong>INVERT</strong></td>
<td>Invert the polarity of the serial pins (normally not needed when level converter, such as the MAX232). May not be used with the internal UART.</td>
</tr>
<tr>
<td><strong>PARITY=X</strong></td>
<td>Where x is N, E, or O.</td>
</tr>
<tr>
<td><strong>BITS=X</strong></td>
<td>Where x is 5-9 (5-7 may not be used with the SCI).</td>
</tr>
<tr>
<td><strong>FLOAT_HIGH</strong></td>
<td>The line is not driven high. This is used for open collector outputs. Bit 6 in RS232_ERRORS is set if the pin is not high at the end of the bit time.</td>
</tr>
<tr>
<td><strong>ERRORS</strong></td>
<td>Used to cause the compiler to keep receive errors in the variable RS232_ERRORS and to reset errors when they occur, and RS232_BUFFER_ERRORS when transmit</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SAMPLE_EARLY</td>
<td>A getc() normally samples data in the middle of a bit time. This option causes the sample to be at the start of a bit time. May not be used with the UART.</td>
</tr>
<tr>
<td>RETURN=pin</td>
<td>For FLOAT_HIGH and MULTI_MASTER this is the pin used to read the signal back. The default for FLOAT_HIGH is the XMIT pin and for MULTI_MASTER the RCV pin.</td>
</tr>
<tr>
<td>MULTI_MASTER</td>
<td>Uses the RETURN pin to determine if another master on the bus is transmitting at the same time. If a collision is detected bit 6 is set in RS232_ERRORS and all future PUTC's are ignored until bit 6 is cleared. The signal is checked at the start and end of a bit time. May not be used with the UART.</td>
</tr>
<tr>
<td>LONG_DATA</td>
<td>Makes getc() return an int16 and putc accept an int16. This is for 9 bit data formats.</td>
</tr>
<tr>
<td>DISABLE_INTS</td>
<td>Will cause interrupts to be disabled when the routines get or put a character. This prevents character distortion for software implemented I/O and prevents interaction between I/O in interrupt handlers and the main program when using the UART.</td>
</tr>
<tr>
<td>STOP=X</td>
<td>To set the number of stop bits (default is 1). This works for both UART and non-UART ports.</td>
</tr>
<tr>
<td>TIMEOUT=X</td>
<td>To set the time getc() waits for a byte in milliseconds. If no character comes in within this time the RS232_ERRORS is set to 0 as well as the return value form getc(). This works for both UART and non-UART ports.</td>
</tr>
<tr>
<td>SYNC_SLAVE</td>
<td>Makes the RS232 line a synchronous slave, making the receive pin a clock in, and the</td>
</tr>
<tr>
<td>PreProcessor</td>
<td>data pin the data in/out.</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td><strong>SYNC_MASTER</strong></td>
<td>Makes the RS232 line a synchronous master, making the receive pin a clock out, and the data pin the data in/out.</td>
</tr>
<tr>
<td><strong>SYNC_MASTER_CONT</strong></td>
<td>Makes the RS232 line a synchronous master mode in continuous receive mode. The receive pin is set as a clock out, and the data pin is set as the data in/out.</td>
</tr>
<tr>
<td><strong>UART1</strong></td>
<td>Sets the XMIT= and RCV= to the chips first hardware UART.</td>
</tr>
<tr>
<td><strong>UART1A</strong></td>
<td>Uses alternate UART pins</td>
</tr>
<tr>
<td><strong>UART2</strong></td>
<td>Sets the XMIT= and RCV= to the chips second hardware UART.</td>
</tr>
<tr>
<td><strong>UART2A</strong></td>
<td>Uses alternate UART pins</td>
</tr>
<tr>
<td><strong>NOINIT</strong></td>
<td>No initialization of the UART peripheral is performed. Useful for dynamic control of the UART baud rate or initializing the peripheral manually at a later point in the program's run time. If this option is used, then setup_uart() needs to be used to initialize the peripheral. Using a serial routine (such as getc() or putc()) before the UART is initialized will cause undefined behavior.</td>
</tr>
<tr>
<td><strong>ICD</strong></td>
<td>Indicates this stream is used to send/receive data through a CCS ICD unit. The default transmit pin is the PIC's ICSPDAT/PGD pin and the default receive pin is the PIC's ICSPCLK/PGC pin. Use XMIT= and RCV= to change the pins used. PCD devices with multiple programming pin pairs, use #device ICSP=x to specify which pin pair ICD it is connected to. Option is not available when Debugging, see DEBUGGER option above.</td>
</tr>
<tr>
<td><strong>UART3</strong></td>
<td>Sets the XMIT= and RCV= to the device's third hardware UART.</td>
</tr>
<tr>
<td><strong>UART4</strong></td>
<td>Sets the XMIT= and RCV= to the device’s</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>fourth hardware UART.</td>
<td>Indicates this stream uses the ICD in a special pass through mode to send/receive serial data to/from PC. The ICSP clock line is the PIC’s receive pin, usually pin B6, and the ICSP data line is the PIC’s transmit pin, usually pin B7.</td>
</tr>
<tr>
<td>MAX_ERROR=x</td>
<td>Specifies the max error percentage the compiler can set the RS232 baud rate from the specified baud before generating an error. Defaults to 3% if not specified.</td>
</tr>
<tr>
<td>Serial Buffer Options:</td>
<td></td>
</tr>
<tr>
<td>RECEIVE_BUFFER=x</td>
<td>Size in bytes of UART circular receive buffer, default if not specified is zero. Uses an interrupt to receive data, supports RDA interrupt or external interrupts.</td>
</tr>
<tr>
<td>TXISR</td>
<td>If TRANSMIT_BUFFER is greater then zero specifies using TBE interrupt for transmitting data. Default is NOTXISR if TXISR or NOTXISR is not specified. TXISR option can only be used when using hardware UART.</td>
</tr>
<tr>
<td>NOTXISR</td>
<td>If TRANSMIT_BUFFER is greater then zero specifies to not use TBE interrupt for transmitting data. Default is NOTXISR if TXISR or NOTXISR is not specified and XMIT_BUFFER is greater then zero</td>
</tr>
<tr>
<td>Flow Control Options:</td>
<td></td>
</tr>
<tr>
<td>RTS = PIN_xx</td>
<td>Pin to use for RTS flow control. When using FLOW_CONTROL_MODE this pin is driven to the active level when it is ready to receive more data. In SIMPLEX_MODE the pin is driven to the active level when it has data to transmit. FLOW_CONTROL_MODE can only be use when using RECEIVE_BUFFER</td>
</tr>
<tr>
<td>RTS_LEVEL=x</td>
<td>Specifies the active level of the RTS pin, HIGH is active high and LOW is active low. Defaults to LOW if not specified.</td>
</tr>
<tr>
<td>CTS = PIN_xx</td>
<td>Pin to use for CTS flow control. In both FLOW_CONTROL_MODE and SIMPLEX_MODE this pin is sampled to see</td>
</tr>
<tr>
<td><strong>PreProcessor</strong></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td></td>
</tr>
</tbody>
</table>

if it clear to send data. If pin is at active level and there is data to send it will send next data byte.

**CTS_LEVEL=x**

Specifies the active level of the CTS pin, HIGH is active high and LOW is active low. Default to LOW if not specified.

**FLOW_CONTROL_MODE**

Specifies how the RTS pin is used. For FLOW_CONTROL_MODE the RTS pin is driven to the active level when ready to receive data. Defaults to FLOW_CONTROL_MODE when neither FLOW_CONTROL_MODE or SIMPLEX_MODE is specified. If RTS pin is not specified then this option is not used.

**SIMPLEX_MODE**

Specifies how the RTS pin is used. For SIMPLEX_MODE the RTS pin is driven to the active level when it has data to send. Defaults to FLOW_CONTROL_MODE when neither FLOW_CONTROL_MODE or SIMPLEX_MODE is specified. If RTS pin is not specified then this option is not used.

**Purpose:**

This directive tells the compiler the baud rate and pins used for serial I/O. This directive takes effect until another RS232 directive is encountered. The #USE DELAY directive must appear before this directive can be used. This directive enables use of built-in functions such as GETC, PUTC, and PRINTF. The functions created with this directive are exported when using multiple compilation units. To access the correct function use the stream identifier.

When using parts with built-in UART and the UART pins are specified, the SCI will be used. If a baud rate cannot be achieved within 3% of the desired value using the current clock rate, an error will be generated. The definition of the RS232_ERRORS is as follows:

No UART:
- Bit 7 is 9th bit for 9 bit data mode (get and put).
- Bit 6 set to one indicates a put failed in float high mode.

With a UART:
- Used only by get:
- Copy of RCSTA register except:
- Bit 0 is used to indicate a parity error.
The definition of the RS232_BUFFER_ERRORS variable is as follows:

- Bit 0 UART Receive overrun error occurred.
- Bit 1 Receive Buffer overflowed.
- Bit 2 Transmit Buffer overflowed.

Warning:
The PIC UART will shut down on overflow (3 characters received by the hardware with a GETC() call). The "ERRORS" option prevents the shutdown by detecting the condition and resetting the UART.

**Examples:**

```
#use rs232(baud=9600, xmit=PIN_A2,rcv=PIN_A3)
```

**Example Files:**

- `ex_cust.c`

**Also See:**

- `getc()`, `putc()`, `printf()`, `setup_uart()`, `RS2322 I/O overview`, `kbhit()`, `puts()`, `putc_send()`, `rcv_buffer_bytes()`, `tx_buffer_bytes()`, `rcv_buffer_full()`, `tx_buffer_full()`, `tx_buffer_available()`

---

**#use rtos**

(The RTOS is only included with the PCW and PCWH packages.)

The CCS Real Time Operating System (RTOS) allows a PIC micro controller to run regularly scheduled tasks without the need for interrupts. This is accomplished by a function (RTOS_RUN()) that acts as a dispatcher. When a task is scheduled to run, the dispatch function gives control of the processor to that task. When the task is done executing or does not need the processor anymore, control of the processor is returned to the dispatch function which then will give control of the processor to the next task that is scheduled to execute at the appropriate time. This process is called cooperative multi-tasking.

**Syntax:**

```
#USE RTOS (options)
```

**Elements:**

<table>
<thead>
<tr>
<th>elements</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timer=X</td>
<td>Where x is 0-4 specifying the timer used by the RTOS.</td>
</tr>
<tr>
<td>minor_cycle=time</td>
<td>Where time is a number followed by s, ms, us, ns. This is the longest time any task will run. Each task's execution rate must be a multiple</td>
</tr>
</tbody>
</table>
PreProcessor

| Purpose: | This directive tells the compiler which timer on the PIC to use for monitoring and when to grant control to a task. Changes to the specified timer’s prescaler will effect the rate at which tasks are executed.

This directive can also be used to specify the longest time that a task will ever take to execute with the minor_cycle option. This simply forces all task execution rates to be a multiple of the minor_cycle before the project will compile successfully. If the this option is not specified the compiler will use a minor_cycle value that is the smallest possible factor of the execution rates of the RTOS tasks.

If the statistics option is specified then the compiler will keep track of the minimum processor time taken by one execution of each task, the maximum processor time taken by one execution of each task, and the total processor time used by each task.

When linking multiple compilation units, this directive must appear exactly the same in each compilation unit.

| Examples: | #use rtos(timer=0, minor_cycle=20ms) |
| Also See: | #TASK |

#use spi

**Syntax:**
```
#USE SPI (options)
```

**Elements:**

<table>
<thead>
<tr>
<th>Options</th>
<th>Set the device as the master. (default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASTER</td>
<td></td>
</tr>
<tr>
<td>SLAVE</td>
<td>Set the device as the slave.</td>
</tr>
<tr>
<td>BAUD=n</td>
<td>Target bits per second, default is as fast as possible.</td>
</tr>
<tr>
<td>CLOCK_HIGH=n</td>
<td>High time of clock in us (not needed if BAUD= is used). (default=0)</td>
</tr>
<tr>
<td>CLOCK_LOW=n</td>
<td>Low time of clock in us (not needed if BAUD= is used). (default=0)</td>
</tr>
<tr>
<td><strong>DI=pin</strong></td>
<td>Optional pin for incoming data.</td>
</tr>
<tr>
<td><strong>DO=pin</strong></td>
<td>Optional pin for outgoing data.</td>
</tr>
<tr>
<td><strong>CLK=pin</strong></td>
<td>Clock pin.</td>
</tr>
<tr>
<td><strong>MODE=n</strong></td>
<td>The mode to put the SPI bus.</td>
</tr>
<tr>
<td><strong>ENABLE=pin</strong></td>
<td>Optional pin to be active during data transfer.</td>
</tr>
<tr>
<td><strong>LOAD=pin</strong></td>
<td>Optional pin to be pulsed active after data is transferred.</td>
</tr>
<tr>
<td><strong>DIAGNOSTIC=pin</strong></td>
<td>Optional pin to the set high when data is sampled.</td>
</tr>
<tr>
<td><strong>SAMPLE_RISE</strong></td>
<td>Sample on rising edge.</td>
</tr>
<tr>
<td><strong>SAMPLE_FALL</strong></td>
<td>Sample on falling edge (default).</td>
</tr>
<tr>
<td><strong>BITS=n</strong></td>
<td>Max number of bits in a transfer. (default=32)</td>
</tr>
<tr>
<td><strong>SAMPLE_COUNT=n</strong></td>
<td>Number of samples to take (uses majority vote). (default=1)</td>
</tr>
<tr>
<td><strong>LOAD_ACTIVE=n</strong></td>
<td>Active state for LOAD pin (0, 1).</td>
</tr>
<tr>
<td><strong>ENABLE_ACTIVE=n</strong></td>
<td>Active state for ENABLE pin (0, 1). (default=0)</td>
</tr>
<tr>
<td><strong>IDLE=n</strong></td>
<td>Inactive state for CLK pin (0, 1). (default=0)</td>
</tr>
<tr>
<td><strong>ENABLE_DELAY=n</strong></td>
<td>Time in us to delay after ENABLE is activated. (default=0)</td>
</tr>
<tr>
<td><strong>DATA_HOLD=n</strong></td>
<td>Time between data change and clock change</td>
</tr>
<tr>
<td><strong>LSB_FIRST</strong></td>
<td>LSB is sent first.</td>
</tr>
<tr>
<td><strong>MSB_FIRST</strong></td>
<td>MSB is sent first. (default)</td>
</tr>
<tr>
<td><strong>STREAM=id</strong></td>
<td>Specify a stream name for this protocol.</td>
</tr>
<tr>
<td><strong>SPI1</strong></td>
<td>Use the hardware pins for SPI Port 1</td>
</tr>
<tr>
<td><strong>SPI2</strong></td>
<td>Use the hardware pins for SPI Port 2</td>
</tr>
<tr>
<td><strong>FORCE_SW</strong></td>
<td>Use a software implementation even when hardware pins are specified</td>
</tr>
<tr>
<td><strong>FORCE_HW</strong></td>
<td>Use the pic hardware SPI.</td>
</tr>
<tr>
<td><strong>SPI3</strong></td>
<td>Use the hardware pins for SPI Port 3</td>
</tr>
<tr>
<td><strong>SPI4</strong></td>
<td>Use the hardware pins for SPI Port 4</td>
</tr>
<tr>
<td><strong>NOINIT</strong></td>
<td>Do not initialize the hardware SPI Port</td>
</tr>
<tr>
<td><strong>XFER16</strong></td>
<td>Uses 16 BIT transfers instead of two 8 BIT transfers</td>
</tr>
</tbody>
</table>

**Purpose:**

The SPI library contains functions to implement an SPI bus. After setting all of the proper parameters in #USE SPI, the spi_xfer() function can be used to both transfer and receive data on the SPI bus.

The SPI1 and SPI2 options will use the SPI hardware onboard the PIC. The most common pins present on hardware SPI are: DI, DO, and CLK. These pins don’t need to be assigned values through the options; the compiler will automatically assign hardware-specific values to these pins. Consult your PIC’s data sheet as to where the pins for hardware SPI are. If hardware SPI is not used, then software SPI will be used. Software SPI is much slower than hardware SPI, but software SPI can use any pins to transfer and receive data.
other than just the pins tied to the PIC's hardware SPI pins.

The MODE option is more or less a quick way to specify how the stream is going to sample data. MODE=0 sets IDLE=0 and SAMPLE_RISE. MODE=1 sets IDLE=0 and SAMPLE_FALL. MODE=2 sets IDLE=1 and SAMPLE_FALL. MODE=3 sets IDLE=1 and SAMPLE_RISE. There are only these 4 MODEs.

SPI cannot use the same pins for DI and DO. If needed, specify two streams: one to send data and another to receive data.

The pins must be specified with DI, DO, CLK or SPIx, all other options are defaulted as indicated above.

Examples:

```plaintext
#use_spi(DI=PIN_B1, DO=PIN_B0, CLK=PIN_B2, ENABLE=PIN_B4, BITS=16)
// uses software SPI
#use_spi(FORCE_HW, BITS=16, stream=SPI_STREAM)
// uses hardware SPI and gives this stream the name SPI_STREAM
```

Example

Files: None

Also See: `spi_xfer()`

---

### `#use standard_io`

**Syntax:**

```
#USE STANDARD_IO (port)
```

**Elements:**

`port` is A, B, C, D, E, F, G, H, J or ALL

**Purpose:**

This directive affects how the compiler will generate code for input and output instructions that follow. This directive takes effect until another `#USE XXX_IO` directive is encountered. The standard method of doing I/O will cause the compiler to generate code to make an I/O pin either input or output every time it is used. On the 5X processors this requires one byte of RAM for every port set to standard I/O.

Standard_io is the default I/O method for all ports.

When linking multiple compilation units be aware this directive only applies to the current compilation unit.
### #use timer

**Syntax:**  

```
#USE TIMER (options)
```

**Elements:**

- **TIMER=x**
  Sets the timer to use as the tick timer. x is a valid timer that the PIC has. Default value is 1 for Timer 1.

- **TICK=xx**
  Sets the desired time for 1 tick. xx can be used with ns(nanoseconds), us (microseconds), ms (milliseconds), or s (seconds). If the desired tick time can't be achieved it will set the time to closest achievable time and will generate a warning specifying the exact tick time. The default value is 1us.

- **BITS=x**
  Sets the variable size used by the get_ticks() and set_ticks() functions for returning and setting the tick time. x can be 8 for 8 bits, 16 for 16 bits, 32 for 32bits or 64 for 64 bits. The default is 32 for 32 bits.

- **ISR**
  Uses the timer's interrupt to increment the upper bits of the tick timer. This mode requires the the global interrupt be enabled in the main program.

- **NOISR**
  The get_ticks() function increments the upper bits of the tick timer. This requires that the get_ticks() function be called more often then the timer's overflow rate. NOISR is the default mode of operation.

- **STREAM=id**
  Associates a stream identifier with the tick timer. The identifier may be used in functions like get_ticks().

---

**Examples:**  

```
#use standard_io(A)
```

**Example Files:**  

ex_cust.c

**Also See:**  

#USE FAST_IO, #USE FIXED_IO, General Purpose I/O
<table>
<thead>
<tr>
<th><strong>DEFINE</strong>=id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creates a define named id which specifies the number of ticks that will occur in one second. Default define name if not specified is TICKS_PER_SECOND. Define name must start with an ASCII letter 'A' to 'Z', an ASCII letter 'a' to 'z' or an ASCII underscore ('_').</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>COUNTER</strong> or <strong>COUNTER</strong>=x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets up specified timer as a counter instead of timer. x specifies the prescaller to setup counter with, default is 1 if x is not specified specified. The function get_ticks() will return the current count and the function set_ticks() can be used to set count to a specific starting value or to clear counter.</td>
</tr>
</tbody>
</table>

**Purpose:** This directive creates a tick timer using one of the PIC's timers. The tick timer is initialized to zero at program start. This directive also creates the define TICKS_PER_SECOND as a floating point number, which specifies that number of ticks that will occur in one second.

**Examples:**

```c
#USE TIMER(TIMER=1,TICK=1ms,BITS=16,NOISR)

unsigned int16 tick_difference(unsigned int16 current, unsigned int16 previous) {
  return(current - previous);
}

void main(void) {
  unsigned int16 current_tick, previous_tick;
  current_tick = previous_tick = get_ticks();
  while(TRUE) {
    current_tick = get_ticks();
    if(tick_difference(current_tick, previous_tick) > 1000) {
      output_toggle(PIN_B0);
    }
    previous_tick = current_tick;
  }
}
```

**Example Files:** None

**Also See:** get_ticks(), set_ticks()
#use touchpad

Syntax:  #USE TOUCHPAD (options)

Elements:  

**RANGE=x**  
Sets the oscillator charge/discharge current range. If x is L, current is nominally 0.1 microamps. If x is M, current is nominally 1.2 microamps. If x is H, current is nominally 18 microamps. Default value is H (18 microamps).

**THRESHOLD=x**  
x is a number between 1-100 and represents the percent reduction in the nominal frequency that will generate a valid key press in software. Default value is 6%.

**SCANTIME=xxMS**  
xx is the number of milliseconds used by the microprocessor to scan for one key press. If utilizing multiple touch pads, each pad will use xx milliseconds to scan for one key press. Default is 32ms.

**PIN=char**  
If a valid key press is determined on “PIN”, the software will return the character “char” in the function touchpad_getc(). (Example: PIN_B0='A')

**SOURCETIME=xxus** (CTMU only)  
xx is the number of microseconds each pin is sampled for by ADC during each scan time period. Default is 10us.

Purpose:  
This directive will tell the compiler to initialize and activate the Capacitive Sensing Module (CSM) or Charge Time Measurement Unit (CTMU) on the microcontroller. The compiler requires use of the TIMER0 and TIMER1 modules for CSM and Timer1 ADC modules for CTMU, and global interrupts must still be activated in the main program in order for the CSM or CTMU to begin normal operation. For most applications, a higher RANGE, lower THRESHOLD, and higher SCANTIME will result better key press detection. Multiple PIN's may be declared in “options”, but they must be valid pins used by the CSM or CTMU. The user may also generate a TIMER0 ISR with TIMER0's interrupt occurring every SCANTIME milliseconds. In this case, the CSM's or CTMU's ISR will be executed first.

Examples:  

```c
#USE TOUCHPAD (THRESHOLD=5, PIN_D5='5', PIN_B0='C')
void main(void){
  char c;
  enable_interruptions(GLOBAL);
```


while(1){
    c = TOUCHPAD_GETC(); //will wait until a pin is detected
}
    //if PIN_B0 is pressed, c will have
    'C'
}                   //if PIN_D5 is pressed, c will have
    '5'

Example        None
Files:         Also See:   touchpad_state(), touchpad_getc(), touchpad_hit()

#warning

Syntax:       #WARNING text
Elements:    text is optional and may be any text
Purpose:      Forces the compiler to generate a warning at the location this directive
               appears in the file. The text may include macros that will be expanded for the
               display. This may be used to see the macro expansion. The command may
               also be used to alert the user to an invalid compile time situation.
               
               To prevent the warning from being counted as a warning, use this syntax:
               #warning/information text

Examples:    #if BUFFER_SIZE < 32
               #warning Buffer Overflow may occur
               #endif

Example      ex_psp.c
Files:       Also See:   #ERROR

#word

Syntax:       #WORD id = x
Elements:  
- *id* is a valid C identifier,
- *x* is a C variable or a constant

Purpose:  
If the id is already known as a C variable then this will locate the variable at address x. In this case the variable type does not change from the original definition. If the id is not known a new C variable is created and placed at address x with the type int16.

Warning: In both cases memory at x is not exclusive to this variable. Other variables may be located at the same location. In fact when x is a variable, then id and x share the same memory location.

Examples:  
```c
#word data = 0x0860

struct
{
  short C;
  short Z;
  short OV;
  short N;
  short RA;
  short IPL0;
  short IPL1;
  short IPL2;
  int upperByte : 8;
} status_register;
#word status_register = 0x42
...
short zero = status_register.Z;
```

Example
Files:  
None

Also See:  
#bit, #byte, #locate, #reserve, Named Registers, Type Specifiers, Type Qualifiers, Enumerated Types, Structures & Unions, Typedef

---

#zero_ram

Syntax:  
```
#ZERO_RAM
```

Elements:  
None

Purpose:  
This directive zero's out all of the internal registers that may be used to hold
variables before program execution begins.

**Examples:**

```c
#zero_ram
void main() {
}
```

**Example Files:**

`ex_cust.c`

**Also See:** None
The CCS compiler provides a lot of built-in functions to access and use the PIC microcontroller's peripherals. This makes it very easy for the users to configure and use the peripherals without going into in depth details of the registers associated with the functionality. The functions categorized by the peripherals associated with them are listed on the next page. Click on the function name to get a complete description and parameter and return value descriptions.

**abs( )**

<table>
<thead>
<tr>
<th>Syntax:</th>
<th>value = abs(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters:</td>
<td>x is any integer or float type.</td>
</tr>
<tr>
<td>Returns:</td>
<td>Same type as the parameter.</td>
</tr>
<tr>
<td>Function:</td>
<td>Computes the absolute value of a number.</td>
</tr>
<tr>
<td>Availability:</td>
<td>All devices</td>
</tr>
<tr>
<td>Requires:</td>
<td>#INCLUDE &lt;stdlib.h&gt;</td>
</tr>
</tbody>
</table>
| Examples:        | signed int target, actual;  
 |                  | ...  
 |                  | error = abs(target-actual); |
| Example Files:   | None |
| Also See:        | labs() |
**sin( )**  **cos( )**  **tan( )**  **asin( )**  **acos()**  **atan()**  **sinh()**  **cosh()**  **tanh()**  **atan2()**

**Syntax:**

```plaintext
val = sin (rad)
val = cos (rad)
val = tan (rad)
rad = asin (val)
rad1 = acos (val)
rad = atan (val)
rad2 = atan2(val, val)
result = sinh(value)
result = cosh(value)
result = tanh(value)
```

**Parameters:**

- `rad` is any float type representing an angle in Radians -2pi to 2pi.
- `val` is any float type with the range -1.0 to 1.0.
- `Value` is any float type

**Returns:**

- `rad` is a float with a precision equal to `val` representing an angle in Radians -pi/2 to pi/2
- `val` is a float with a precision equal to `rad` within the range -1.0 to 1.0.
- `rad1` is a float with a precision equal to `val` representing an angle in Radians 0 to pi
- `rad2` is a float with a precision equal to `val` representing an angle in Radians -pi to pi
- Result is a float with a precision equal to `value`

**Function:**

These functions perform basic Trigonometric functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sin</strong></td>
<td>returns the sine value of the parameter (measured in radians)</td>
</tr>
<tr>
<td><strong>cos</strong></td>
<td>returns the cosine value of the parameter (measured in radians)</td>
</tr>
<tr>
<td><strong>tan</strong></td>
<td>returns the tangent value of the parameter (measured in radians)</td>
</tr>
<tr>
<td><strong>asin</strong></td>
<td>returns the arc sine value in the range [-pi/2,+pi/2] radians</td>
</tr>
<tr>
<td><strong>acos</strong></td>
<td>returns the arc cosine value in the range [0,pi] radians</td>
</tr>
<tr>
<td><strong>atan</strong></td>
<td>returns the arc tangent value in the range [-pi/2,+pi/2] radians</td>
</tr>
<tr>
<td><strong>atan2</strong></td>
<td>returns the arc tangent of y/x in the range [-pi,+pi] radians</td>
</tr>
<tr>
<td><strong>sinh</strong></td>
<td>returns the hyperbolic sine of x</td>
</tr>
</tbody>
</table>
**Built-in Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cosh</code></td>
<td>returns the hyperbolic cosine of x</td>
</tr>
<tr>
<td><code>tanh</code></td>
<td>returns the hyperbolic tangent of x</td>
</tr>
</tbody>
</table>

Note on error handling:
If "errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Domain error occurs in the following cases:
- `asin`: when the argument not in the range\([-1,+1]\]
- `acos`: when the argument not in the range\([-1,+1]\]
- `atan2`: when both arguments are zero

Range error occur in the following cases:
- `cosh`: when the argument is too large
- `sinh`: when the argument is too large

**Availability:** All devices

**Requires:** 
`#INCLUDE <math.h>`

**Examples:**
```c
float phase;
// Output one sine wave
for(phase=0; phase<2*3.141596; phase+=0.01)
    set_analog_voltages( sin(phase)+1 );
```

**Example Files:** `ex_tank.c`

**Also See:** `log()`, `log10()`, `exp()`, `pow()`, `sqrt()`

---

**adc_done( ) adc_done2( )**

**Syntax:**
```c
value = adc_done();
value = adc_done2( );
value=adc_done([channel])
```

**Parameters:** None
- `channel` is an optional parameter for specifying the channel to check if the conversion is done. If not specified will use channel specified in the last call to `set_adc_channel()`, `read_adc()` or `adc_done()`. Only available for...
dsPIC33EPxxGSxxx family.

Returns: A short int. TRUE if the A/D converter is done with conversion, FALSE if it is still busy.

Function: Can be polled to determine if the A/D has valid data.

Availability: Only available on devices with built in analog to digital converters

Requires: None

Examples:

```c
int16 value;
setup_adc_ports(sAN0|sAN1, VSS_VDD);
setup_adc(ADC_CLOCK_DIV_4|ADC_TAD_MUL_8);
set_adc_channel(0);
read_adc(ADC_STARTONLY);

int1 done = adc_done();
while(!done) {
    done = adc_done();
}
value = read_adc(ADC_READONLY);
printf("A/C value = %LX\n", value);
```

Example Files: None

Also See: setup_adc(), set_adc_channel(), setup_adc_ports(), read_adc(), ADC Overview

---

### adc_read()

**Syntax:**

\[
\text{result}=\text{adc\_read} (\text{register})
\]

**Parameters:**

- **register** - ADC register to read:
  - ADC_RESULT
  - ADC_ACCUMULATOR
  - ADC_FILTER

**Returns:** int8 or in16 read from the specified register. Return size depends on which register is being read. For example, ADC_RESULT register is 16 bits and ADC_COUNT register is 8-bits.

**Function:** Reads one of the Analog-to-Digital Converter with Computation (ADC2)
Module registers
Availability: All devices with an ADC2 Module
Requires: Constants defined in the device's .h file
Examples: FilteredResult=adc_read(ADC_FILTER);
Also See: ADC Overview, setup_adc(), setup_adc_ports(), set_adc_channel(), read_adc(),
#DEVICE, adc_write(), adc_status(), set_adc_trigger()

### adc_status()

Syntax: status=adc_status()

Parameters: None
Returns: int8 value of the ADSTAT register
Function: Read the current value of the ADSTAT register of the Analog-to-Digital Converter with Computation (ADC2) Module.
Availability: All devices with an ADC2 Module
Requires: Nothing
Examples: while((adc_status() & ADC_UPDATING)==0);
Average=adc_read(ADC_FILTER);

Also See: ADC Overview, setup_adc(), setup_adc_ports(), set_adc_channel(), read_adc(),
#DEVICE, adc_read(), adc_write(), set_adc_trigger()

### adc_write()

Syntax: adc_write(register, value)

Parameters: register - ADC register to write:
  • ADC_REPEAT
  • ADC_SET_POINT
  • ADC_LOWER_THRESHOLD
  • ADC_UPPER_THRESHOLD
Returns: undefined
Function: Write one of the Analog-to-Digital Converter with Computation (ADC2) Module registers.
Availability: All devices with an ADC2 Module
Requires: Constants defined in the device's .h file
Examples: adc_write(ADC_SET_POINT, 300);

Also See: ADC Overview, setup_adc(), setup_adc_ports(), set_adc_channel(),
assert()

Syntax: \texttt{assert (condition);}

Parameters: \texttt{condition} is any relational expression

Returns: Nothing

Function: This function tests the condition and if FALSE will generate an error message on STDERR (by default the first USE RS232 in the program). The error message will include the file and line of the assert(). No code is generated for the assert() if you \#define NODEBUG. In this way you may include asserts in your code for testing and quickly eliminate them from the final program.

Availability: All devices

Requires: assert.h and \#USE RS232

Examples:
\begin{verbatim}
assert( number_of_entries<TABLE_SIZE );
// If number_of_entries is >= TABLE_SIZE then
// the following is output at the RS232:
// Assertion failed, file myfile.c, line 56
\end{verbatim}

Example Files: None

Also See: \#USE RS232, RS232 I/O Overview

atoe

Syntax: \texttt{atoe(string);}

Parameters: \texttt{string} is a pointer to a null terminated string of characters.

Returns: Result is a floating point number
**Function:** Converts the string passed to the function into a floating point representation. If the result cannot be represented, the behavior is undefined. This function also handles E format numbers.

**Availability:** All devices

**Requires:** #INCLUDE <stdlib.h>

**Examples:**

```c
char string [10];
float x;

strcpy (string, "12E3");
x = atoe(string);
// x is now 12000.00
```

**Example Files:** None

**Also See:** atoi(), atol(), atoi32(), atof(), printf()

### Syntax:

```c
result = atof (string)
or
result = atof48(string)
or
result=atof64(string)
or
result=strtof48(string)
```

**Parameters:** *string* is a pointer to a null terminated string of characters.

**Returns:** Result is a floating point number in single, extended or double precision format

**Function:** Converts the string passed to the function into a floating point representation. If the result cannot be represented, the behavior is undefined.

**Availability:** All devices

**Requires:** #INCLUDE <stdlib.h>

**Examples:**

```c
char string [10];
float x;
```
```c
strcpy (string, "123.456");
x = atof(string);
// x is now 123.456
```

**Example Files:**  
`ex_tank.c`

**Also See:**  
`atoi()`, `atol()`, `atoi32()`, `printf()`

### pin_select()

**Syntax:**
```
pin_select(peripheral_pin, pin, [unlock],[lock])
```

**Parameters:**
- `peripheral_pin` – a constant string specifying which peripheral pin to map the specified pin to. Refer to `#pin_select` for all available strings. Using “NULL” for the peripheral_pin parameter will unassign the output peripheral pin that is currently assigned to the pin passed for the pin parameter.
  
  `pin` – the pin to map to the specified peripheral pin. Refer to device’s header file for pin defines. If the peripheral_pin parameter is an input, passing FALSE for the pin parameter will unassign the pin that is currently assigned to that peripheral pin.
  
  `unlock` – optional parameter specifying whether to perform an unlock sequence before writing the RPINRx or RPORx register determined by peripheral_pin and pin options. Default is TRUE if not specified. The unlock sequence must be performed to allow writes to the RPINRx and RPORx registers. This option allows calling pin_select() multiple times without performing an unlock sequence each time.
  
  `lock` – optional parameter specifying whether to perform a lock sequence after writing the RPINRx or RPORx registers. Default is TRUE if not specified. Although not necessary it is a good idea to lock the RPINRx and RPORx registers from writes after all pins have been mapped. This option allows calling `pin_select()` multiple times without performing a lock sequence each time.

**Returns:**
Nothing.

**Availability:** On device with remappable peripheral pins.

**Requires:** Pin defines in device’s header file.

**Examples:**
```
pin_select("U2TX",PIN_B0);
```
//Maps PIN_B0 to U2TX //peripheral pin, performs unlock //and lock sequences.

pin_select("U2TX",PIN_B0,TRUE,FALSE);

//Maps PIN_B0 to U2TX //peripheral pin and performs //unlock sequence.

pin_select("U2RX",PIN_B1,FALSE,TRUE);

//Maps PIN_B1 to U2RX //peripheral pin and performs lock //sequence.

Example Files: None.
Also See: #pin_select

### atoi()  atol()  atoi32()  atol32()
atoi48()  atoi64()

**Syntax:**

```c
ivalue = atoi(string)
or
lvalue = atol(string)
or
i32value = atoi32(string)
or
i48value=atoi48(string)
or
i64value=atoi64(string)
or
L32value=atol32(string)
```

**Parameters:**

`string` is a pointer to a null terminated string of characters.

**Returns:**

- `ivalue` is an 8 bit int.
- `lvalue` is a 16 bit int.
- `i32value` is a 32 bit int.
- `48value` is a 48 bit int.
- `i64value` is a 64 bit int.
- `L32value` is a 32 bit long.

**Function:** Converts the string passed to the function into an int representation. Accepts both decimal and hexadecimal argument. If the
result cannot be represented, the behavior is undefined.

<table>
<thead>
<tr>
<th>Availability:</th>
<th>All devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires:</td>
<td>#INCLUDE &lt;stdlib.h&gt;</td>
</tr>
</tbody>
</table>

**Examples:**

```c
char string[10];
int x;

strcpy(string,"123");
x = atoi(string);
// x is now 123
```

**Example Files:**

*input.c*

**Also See:**

printf()

---

### at_clear_interrupts()

**Syntax:**

```c
at_clear_interrupts(interrupts);
```

**Parameters:**

<table>
<thead>
<tr>
<th>interrupts - an 8-bit constant specifying which AT interrupts to disable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The constants are defined in the device's header file as:</td>
</tr>
<tr>
<td>· AT_PHASE_INTERRUPT</td>
</tr>
<tr>
<td>· AT_MISSING_PULSE_INTERRUPT</td>
</tr>
<tr>
<td>· AT_PERIOD_INTERRUPT</td>
</tr>
<tr>
<td>· AT_CC3_INTERRUPT</td>
</tr>
<tr>
<td>· AT_CC2_INTERRUPT</td>
</tr>
<tr>
<td>· AT_CC1_INTERRUPT</td>
</tr>
</tbody>
</table>

**Returns:**

Nothing

**Function:**

To disable the Angular Timer interrupt flags. More than one interrupt can be cleared at a time by or'ing multiple constants together in a single call, or calling function multiple times for each interrupt to clear.

**Availability:**

All devices with an AT module.

**Requires:**

Constants defined in the device's header file

**Examples:**

#INT-AT1
void1_isr(void)
{
    if(at_interrupt_active(AT_PERIOD_INTERRUPT))
    {
        handle_period_interrupt();
        at_clear_interrupts(AT_PERIOD_INTERRUPT);
    }
    if(at_interrupt_active(AT_PHASE_INTERRUPT))
    {
        handle_phase_interrupt();
        at_clear_interrupts(AT_PHASE_INTERRUPT);
    }
}

Example Files: None

Also See: at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

at_disable_interrupts()

Syntax: at_disable_interrupts(interrupts);

Parameters: interrupts - an 8-bit constant specifying which AT interrupts to disable. The constants are defined in the device's header file as:

- AT_PHASE_INTERRUPT
- AT_MISSING_PULSE_INTERRUPT
- AT_PERIOD_INTERRUPT
- AT_CC3_INTERRUPT
- AT_CC2_INTERRUPT
- AT_CC1_INTERRUPT

Returns: Nothing

Function: To disable the Angular Timer interrupts. More than one interrupt can be disabled at a time by or'ing multiple constants together in a single call, or calling function multiple times for each interrupt to be disabled.
### at_enable_interruptions()

**Syntax:**

```
at_enable_interruptions(interrupts);
```

**Parameters:**

- `interrupts` - an 8-bit constant specifying which AT interrupts to enable. The constants are defined in the device's header file as:
  - AT_PHASE_INTERRUPT
  - AT_MISSING_PULSE_INTERRUPT
  - AT_PERIOD_INTERRUPT
  - AT_CC3_INTERRUPT
  - AT_CC2_INTERRUPT
  - AT_CC1_INTERRUPT

**Returns:**

Nothing

**Function:**

To enable the Angular Timer interrupts. More than one interrupt can be enabled at a time by or'ing multiple constants together in a single call, or calling function multiple times for each interrupt to be enabled.

**Availability:**

All devices with an AT module.

**Requires:**

Constants defined in the device's header file

**Examples:**

```
at_enable_interruptions(AT_PHASE_INTERRUPT);
at_enable_interruptions(AT_PERIOD_INTERRUPT|AT_CC1_INTERRUPT);
```

**Example Files:**

None
Built-in Functions

Also See: setup_at(), at_set_resolution(), at_get_resolution(),
at_set_missing_pulse_delay(), at_get_missing_pulse_delay(),
at_get_phase_counter(), at_set_set_point(), at_get_set_point(),
at_get_set_point(), at_get_set_point_error(), at_disable_interrupts(),
at_clear_interrupts(), at_interrupt_active(), at_setup_cc(),
at_set_compare_time(), at_get_capture(), at_get_status()

at_get_capture()  

Syntax: result=at_get_capture(which);;

Parameters: which - an 8-bit constant specifying which AT Capture/Compare module to get the capture time from, can be 1, 2 or 3.

Returns: A 16-bit integer

Function: To get one of the Angular Timer Capture/Compare modules capture time.

Availability: All devices with an AT module.

Requires: Nothing

Examples: result1=at_get_capture(1);
result2=at_get_capture(2);

Example Files: None

Also See: setup_at(), at_set_resolution(), at_get_resolution(),
at_set_missing_pulse_delay(), at_get_missing_pulse_delay(),
at_get_phase_counter(), at_set_set_point(), at_get_set_point(),
at_get_set_point(), at_get_set_point_error(), at_disable_interrupts(),
at_clear_interrupts(), at_interrupt_active(), at_setup_cc(),
at_set_compare_time(), at_get_capture(), at_get_status()
### at_get_missing_pulse_delay( )

**Syntax:**

\[
\text{result} = \text{at\_get\_missing\_pulse\_delay}();
\]

**Parameters:** None.

**Returns:** A 16-bit integer

**Function:** To setup the Angular Timer Missing Pulse Delay

**Availability:** All devices with an AT module.

**Requires:** Nothing

**Examples:**

\[
\text{result} = \text{at\_get\_missing\_pulse\_delay}();
\]

**Example Files:** None

**Also See:**

- at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrups(), at_disable_interrups(), at_clear_interruptions(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

### at_get_period( )

**Syntax:**

\[
\text{result} = \text{at\_get\_period}();
\]

**Parameters:** None.

**Returns:** A 16-bit integer. The MSB of the returned value specifies whether the period counter rolled over one or more times. 1 - counter rolled over at least once, 0 - value returned is valid.

**Function:** To get Angular Timer Measured Period

**Availability:** All devices with an AT module.

**Requires:** Nothing
Examples: `result=at_get_period();`

Example Files: None

Also See: `at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_set_phase_counter(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

---

**at_get_phase_counter()**

Syntax: `result=at_get_phase_counter();`

Parameters: None.

Returns: A 16-bit integer.

Function: To get the Angular Timer Phase Counter

Availability: All devices with an AT module.

Requires: Nothing

Examples: `result=at_get_phase_counter();`

Example Files: None

Also See: `at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_set_phase_counter(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()"
**at_get_resolution()**

Syntax: `result=at_get_resolution();`

Parameters: None

Returns: A 16-bit integer

Function: To setup the Angular Timer Resolution

Availability: All devices with an AT module.

Requires: Nothing

Examples:

```
result=at_get_resolution();
```

Example Files: None

Also See:
- `at_set_resolution()`, `at_set_missing_pulse_delay()`,
- `at_get_missing_pulse_delay()`, `at_get_period()`, `at_get_phase_counter()`,
- `at_set_set_point()`, `at_get_set_point()`, `at_get_set_point_error()`,
- `at_enable_interrupts()`, `at_disable_interrupts()`, `at_clear_interrupts()`,
- `at_interrupt_active()`, `at_setup_cc()`, `at_set_compare_time()`,
- `at_get_capture()`, `at_get_status()`, `setup_at()`

---

**at_get_set_point()**

Syntax: `result=at_get_set_point();`

Parameters: None

Returns: A 16-bit integer

Function: To get the Angular Timer Set Point

Availability: All devices with an AT module.

Requires: Nothing

Examples:

```
result=at_get_set_point();
```
Example Files: None

Also See: \at_set_resolution(), \at_get_resolution(), \at_set_missing_pulse_delay(), \at_get_missing_pulse_delay(), \at_get_period(), \at_get_phase_counter(), \at_set_set_point(), \at_get_set_point_error(), \at_enable_interrupts(), \at_disable_interrupts(), \at_clear_interrupts(), \at_interrupt_active(), \at_setup_cc(), \at_set_compare_time(), \at_get_capture(), \at_get_status(), \setup_at()

---

**at_get_set_point_error()**

**Syntax:**

\begin{verbatim}
result=at_get_set_point_error();
\end{verbatim}

**Parameters:** None

**Returns:** A 16-bit integer

**Function:** To get the Angular Timer Set Point Error, the error of the measured period value compared to the threshold setting.

**Availability:** All devices with an AT module.

**Requires:** Nothing

**Examples:**

\begin{verbatim}
result=at_get_set_point_error();
\end{verbatim}

**Example Files:** None

**Also See:** \at_set_resolution(), \at_get_resolution(), \at_set_missing_pulse_delay(), \at_get_missing_pulse_delay(), \at_get_period(), \at_get_phase_counter(), \at_set_set_point(), \at_get_set_point_error(), \at_enable_interrupts(), \at_disable_interrupts(), \at_clear_interrupts(), \at_interrupt_active(), \at_setup_cc(), \at_set_compare_time(), \at_get_capture(), \at_get_status(), \setup_at()
at_get_status( )

Syntax: result = at_get_status();

Parameters: None

Returns: An 8-bit integer. The possible results are defined in the device's header file as:
· AT_STATUS_PERIOD_AND_PHASE_VALID
· AT_STATUS_PERIOD_LESS_THAN_PREVIOUS

Function: To get the status of the Angular Timer module.

Availability: All devices with an AT module.

Requires: Nothing

Examples: if((at_get_status() & AT_STATUS_PERIOD_AND_PHASE_VALID) == AT_STATUS_PERIOD_AND_PHASE_VALID)
[  Period = at_get_period();
  Phase = at_get_phase();
]

Example Files: None

Also See: at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(),
at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(),
at_set_set_point(), at_get_set_point(), at_get_set_point_error(),
at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(),
at_interrupt_active(), at_setup_cc(), at_set_compare_time(),
at_get_capture(), setup_at()

at_interrupt_active( )

Syntax: result = at_interrupt_active(interrupt);

Parameters: interrupts - an 8-bit constant specifying which AT interrupts to check if its flag is set. The constants are defined in the device's header file as:
· AT_PHASE_INTERRUPT
· AT_MISSING_PULSE_INTERRUPT
at_set_compare_time( )

Syntax: `at_set_compare_time(which, compare_time);`

Parameters: `which` - an 8-bit constant specifying which AT Capture/Compare module to set the compare time for, can be 1, 2, or 3.
**compare_time** - a 16-bit constant or variable specifying the value to trigger an interrupt/output pulse.

**Returns:** Nothing

**Function:** To set one of the Angular Timer Capture/Compare module's compare time.

**Availability:** All devices with an AT module.

**Requires:** Constants defined in the device's header file

**Examples:**
- `at_set_compare_time(1, 0x1FF);`
- `at_set_compare_time(3, compare_time);`

**Example Files:** None

**Also See:** `at_set_resolution()`, `at_get_resolution()`, `at_set_missing_pulse_delay()`, `at_get_missing_pulse_delay()`, `at_get_period()`, `at_get_phase_counter()`, `at_set_set_point()`, `at_get_set_point()`, `at_get_set_point_error()`, `at_enable_interrupts()`, `at_disable_interrupts()`, `at_clear_interrupts()`, `at_interrupt_active()`, `at_setup_cc()`, `at_get_capture()`, `at_get_status()`, `setup_at()`

---

**at_set_missing_pulse_delay()**

**Syntax:**

```
at_set_missing_pulse_delay(pulse_delay);
```

**Parameters:**

- `pulse_delay` - a signed 16-bit constant or variable to set the missing pulse delay.

**Returns:** Nothing

**Function:** To setup the Angular Timer Missing Pulse Delay

**Availability:** All devices with an AT module.

**Requires:** Nothing

**Examples:**
- `at_set_missing_pulse_delay(pulse_delay);`
Built-in Functions

Example Files: None

Also See: at_set_resolution(), at_get_resolution(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

at_set_resolution()

Syntax: at_set_resolution(resolution);

Parameters: resolution - a 16-bit constant or variable to set the resolution.

Returns: Nothing

Function: To setup the Angular Timer Resolution

Availability: All devices with an AT module.

Requires: Nothing

Examples: at_set_resolution(resolution);

Example Files: None

Also See: at_get_resolution(), at_set_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()
### at_set_set_point()

**Syntax:**
```
at_set_set_point(set_point);
```

**Parameters:**
- `set_point` - a 16-bit constant or variable to set the set point. The set point determines the threshold setting that the period is compared against for error calculation.

**Returns:**
Nothing

**Function:**
To get the Angular Timer Set Point

**Availability:**
All devices with an AT module.

**Requires:**
Nothing

**Examples:**
```
at_set_set_point(set_point);
```

**Example Files:**
None

**Also See:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

### at_setup_cc()

**Syntax:**
```
at_setup_cc(which, settings);
```

**Parameters:**
- `which` - an 8-bit constant specifying which AT Capture/Compare to setup, can be 1, 2 or 3.

- `settings` - a 16-bit constant specifying how to setup the specified AT Capture/Compare module. See the device’s header file for all options. Some of the typical options include:
  - AT_CC_ENABLED
  - AT_CC_DISABLED
  - AT_CC_CAPTURE_MODE
Built-in Functions

- AT_CC_COMPARE_MODE
- AT_CAPTURE_FALLING_EDGE
- AT_CAPTURE_RISING_EDGE

Returns: Nothing

Function: To setup one of the Angular Timer Capture/Compare modules to the specified settings.

Availability: All devices with an AT module.

Requires: Constants defined in the device's header file

Examples:

```c
at_setup_cc(1, AT_CC_ENABLED | AT_CC_CAPTURE_MODE | AT_CAPTURE_FALLING_EDGE | AT_CAPTURE_INPUT_ATCAP);
```

```c
at_setup_cc(2, AT_CC_ENABLED | AT_CC_CAPTURE_MODE | AT_CC_ACTIVE_HIGH);
```

Example Files: None

Also See: at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_set_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

---

**bit_clear()**

Syntax: `bit_clear(var, bit)`

Parameters:

- `var` may be any bit variable (any lvalue)
- `bit` is a number 0- 63 representing a bit number, 0 is the least significant bit.

Returns: undefined

Function: Simply clears the specified bit in the given variable. The least significant bit is 0. This function is the similar to: `var &= ~(1<<bit);`

Availability: All devices
### Requires:
Nothing

### Examples:
```c
int x;
x=5;
bit_clear(x,2);
// x is now 1
```

### Example Files:
`ex_patg.c`

### Also See:
`bit_set()`, `bit_test()`

---

## bit_first()

### Syntax:
```
N = bit_first (value, var)
```

### Parameters:
- `value` is a 0 to 1 to be shifted in
- `var` is a 16 bit integer.

### Returns:
An 8 bit integer

### Function:
This function sets `N` to the 0 based position of the first occurrence of `value`. The search starts from the right or least significant bit.

### Availability:
30F/33F/24-bit devices

### Requires:
Nothing

### Examples:
```c
Int16 var = 0x0033;
Int8 N = 0;
// N = 2
N = bit_first (0, var);
```

### Example Files:
None

### Also See:
`shift_right()`, `shift_left()`, `rotate_right()`, `rotate_left()`
**bit_last( )**

Syntax:  

```
N = bit_last (value, var)
N = bit_last(var)
```

Parameters:  

- `value` is a 0 to 1 to search for  
- `var` is a 16 bit integer.

Returns:  

An 8-bit integer

Function:  

The first function will find the first occurrence of value in the var starting with the most significant bit.  
The second function will note the most significant bit of var and then search for the first different bit.  
Both functions return a 0 based result.

Availability:  

30F/33F/24-bit devices

Requires:  

Nothing

Examples:  

```
//Bit pattern
//11101110 11111111
Int16 var = 0xEEFF;
Int8 N = 0;
//N is assigned 12
N = bit_last (0, var);
//N is assigned 12
N = bit_last(var);
```

Example Files:  

None

Also See:  

`shift_right()`, `shift_left()`, `rotate_right()`, `rotate_left()`

---

**bit_set( )**

Syntax:  

```
bit_set(var, bit)
```

Parameters:  

- `var` may be any variable (any lvalue)  
- `bit` is a number 0-63 representing a bit number, 0 is the least significant bit.

Returns:  

Undefined
**Function:**
Sets the specified bit in the given variable. The least significant bit is 0. This function is the similar to: `var |= (1<<bit);`

**Availability:**
All devices

**Requires:**
Nothing

**Examples:**
```c
int x;
x=5;
bit_set(x,3);
// x is now 13
```

**Example Files:**
`ex_parag.c`

**Also See:**
`bit_clear()`, `bit_test()`

---

**bit_test( )**

**Syntax:**
```c
value = bit_test (var, bit)
```

**Parameters:**
- `var` may be a any bit variable (any lvalue)
- `bit` is a number 0-63 representing a bit number, 0 is the least significant bit.

**Returns:**
0 or 1

**Function:**
Tests the specified bit in the given variable. The least significant bit is 0. This function is much more efficient than, but otherwise similar to: `((var & (1<<bit)) != 0)`

**Availability:**
All devices

**Requires:**
Nothing

**Examples:**
```c
if( bit_test(x,3) || !bit_test (x,1) ) {
    //either bit 3 is 1 or bit 1 is 0
}
```

```c
if(data!=0)
    for(i=31;!bit_test(data, i);i--)
        ;
// i now has the most significant bit in data
// that is set to a 1
```
bsearch( )

**Syntax:**

\[ ip = \text{bsearch} \left (&key, base, num, width, compare \right) \]

**Parameters:**

- **key:** Object to search for
- **base:** Pointer to array of search data
- **num:** Number of elements in search data
- **width:** Width of elements in search data
- **compare:** Function that compares two elements in search data

**Returns:**

bsearch returns a pointer to an occurrence of key in the array pointed to by base. If key is not found, the function returns NULL. If the array is not in order or contains duplicate records with identical keys, the result is unpredictable.

**Function:**

Performs a binary search of a sorted array

**Availability:**

All devices

**Requires:**

```
#include <stdlib.h>
```

**Examples:**

```c
int nums[5]={1,2,3,4,5};
int compar(const void *arg1,const void *arg2);

void main() {
    int *ip, key;
    key = 3;
    ip = bsearch(&key, nums, 5, sizeof(int), compar);
}
```

```c
int compar(const void *arg1,const void *arg2) {
    if ( * (int *) arg1 < ( * (int *) arg2) return -1
    else if ( * (int *) arg1 == ( * (int *) arg2) return 0
    else return 1;
}
```

**Example Files:**

None

**Also See:**

qsort()
**calloc( )**

Syntax: \[ \text{ptr=} \text{calloc(} nmem, \text{ size)} \]

Parameters: \( nmem \) is an integer representing the number of member objects  
\( \text{size} \) is the number of bytes to be allocated for each one of them.

Returns: A pointer to the allocated memory, if any. Returns null otherwise.

Function: The calloc function allocates space for an array of \( nmem \) objects whose size is specified by \( \text{size} \). The space is initialized to all bits zero.

Availability: All devices

Requires: 
\#INCLUDE <stdlib.h>

Examples:
\[
\text{int * iptr;  
iptr=} \text{calloc(}5,10);  
// iptr will point to a block of memory of  
// 50 bytes all initialized to 0.
\]

Example Files: None

Also See: realloc(), free(), malloc()

---

**ceil( )**

Syntax: \( \text{result }= \text{ceil(} \text{value)} \)

Parameters: \( \text{value} \) is any float type

Returns: A float with precision equal to \( \text{value} \)

Function: Computes the smallest integer value greater than the argument. CEIL(12.67) is 13.00.

Availability: All devices

Requires: 
\#INCLUDE<math.h>
Examples:  
// Calculate cost based on weight rounded
// up to the next pound

cost = ceil(weight) * DollarsPerPound;

Example Files:  None

Also See:  floor()

clear_interrupt( )

Syntax:  clear_interrupt(level)

Parameters:  level - a constant defined in the devices.h file

Returns:  undefined

Function:  Clears the interrupt flag for the given level. This function is designed for
use with a specific interrupt, thus eliminating the GLOBAL level as a
possible parameter. Some chips that have interrupt on change for
individual pins allow the pin to be specified like INT_RA1.

Availability:  All devices

Requires:  Nothing

Examples:  clear_interrupt(int_timer1);

Example Files:  None

Also See:  enable_interrupts, #INT, Interrupts Overview
disable_interrupts(), interrupt_active()
clear_pwm1_interrupt( )
clear_pwm2_interrupt( )
clear_pwm3_interrupt( )
clear_pwm4_interrupt( )
clear_pwm5_interrupt( )
clear_pwm6_interrupt( )

Syntax:  
clear_pwm1_interrupt (interrupt)
clear_pwm2_interrupt (interrupt)
clear_pwm3_interrupt (interrupt)
clear_pwm4_interrupt (interrupt)
clear_pwm5_interrupt (interrupt)
clear_pwm6_interrupt (interrupt)

Parameters:  
interrupt - 8-bit constant or variable. Constants are defined in the device's header file as:
- PWM_PERIOD_INTERRUPT
- PWM_DUTY_INTERRUPT
- PWM_PHASE_INTERRUPT
- PWM_OFFSET_INTERRUPT

Returns:  
undefined.

Function:  
Clears one of the above PWM interrupts, multiple interrupts can be cleared by or'ing multiple options together.

Availability:  
Devices with a 16-bit PWM module.

Requires:  
Nothing

Examples:  
clear_pwm1_interrupt(PWM_PERIOD_INTERRUPT);
clear_pwm1_interrupt(PWM_PERIOD_INTERRUPT | PWM_DUTY_INTERRUPT);

Example Files:  
Also See:  
setup_pwm(), set_pwm_duty(), set_pwm_phase(), set_pwm_period(),
set_pwm_offset(), enable_pwm_interrupt(), disable_pwm_interrupt(),
pwm_interrupt_active()
## cog_status()

**Syntax:**
\[
\text{value} = \text{cog\_status}();
\]

**Parameters:**
None

**Returns:**
value - the status of the COG module

**Function:**
To determine if a shutdown event occurred on the Complementary Output Generator (COG) module.

**Availability:**
All devices with a COG module.

**Examples:**
\[
\begin{align*}
\text{if} & (\text{cog\_status}() == \text{COG\_AUTO\_SHUTDOWN}) \\
& \text{cog\_restart}();
\end{align*}
\]

**Example Files:**
None

**Also See:**
setup_cog(), set_cog_dead_band(), set_cog_blanking(), set_cog_phase(), cog_restart()

## cog_restart()

**Syntax:**
\[
\text{cog\_restart}();
\]

**Parameters:**
None

**Returns:**
Nothing

**Function:**
To restart the Complementary Output Generator (COG) module after an auto-shutdown event occurs, when not using auto-restart option of module.

**Availability:**
All devices with a COG module.

**Examples:**
\[
\begin{align*}
\text{if} & (\text{cog\_status}() == \text{COG\_AUTO\_SHUTDOWN}) \\
& \text{cog\_restart}();
\end{align*}
\]

**Example Files:**
None

**Also See:**
setup_cog(), set_cog_dead_band(), set_cog_blanking(), set_cog_phase(), cog_status()
**Syntax:**

```c
Result = crc_calc (data, [width]);
Result = crc_calc (ptr, len, [width]);
Result = crc_calc8 (data, [width]);
Result = crc_calc8 (ptr, len, [width]);
Result = crc_calc16 (data, [width]);       // same as crc_calc ( )
Result = crc_calc16 (ptr, len, [width]);   // same as crc_calc ( )
Result = crc_calc32 (data, [width]);
Result = crc_calc32 (ptr, len, [width]);
```

**Parameters:**

- **data** - This is one double word, word or byte that needs to be processed when using `crc_calc16( )`, or `crc_calc8( )`, `crc_calc32( )`

- **ptr** - is a pointer to one or more double words, words or bytes of data

- **len** - number of double words, words or bytes to process for function calls `crc_calc16( )`, or `crc_calc8( )`, `crc_calc32( )`

- **width** - optional parameter used to specify the input data bit width to use with the functions `crc_calc16( )`, and `crc_calc8( )`, `crc_calc32( )`. Only available on devices with a 32-bit CRC peripheral. If not specified, it defaults to the width of the return bit width, `crc_calc16( )` and 8-bit `crc_calc8( )`. For devices with a 16-bit for CRC the input data bit width is the same as the return bit width, `crc_calc16( )` and 8-bit `crc_calc8( )`.  

**Returns:**

Returns the result of the final CRC calculation.

**Function:**

This will process one data double word, word or byte or **len** double words, words or bytes of data using the CRC engine.

**Availability:**

Only the devices with built in CRC module.

**Requires:**

Nothing

**Examples:**

```c
int16 data[8];
Result = crc_calc(data, 8);
```

**Example Files:**

None
Also See: setup_crc(); crc_init()

**crc_init(mode)**

**Syntax:**
crc_init (data);

**Parameters:**
data - This will setup the initial value used by write CRC shift register. Most commonly, this register is set to 0x0000 for start of a new CRC calculation.

**Returns:**
undefined

**Function:**
Configures the CRCWDAT register with the initial value used for CRC calculations.

**Availability:**
Only the devices with built in CRC module.

**Requires:**
Nothing

**Examples:**
crc_init (); // Starts the CRC accumulator out at 0

crc_init(0xFEEE); // Starts the CRC accumulator out at 0xFEEE

**Example Files:**
None

**Also See:**
setup_crc(), crc_calc(), crc_calc8()

**cwg_status( )**

**Syntax:**
value = cwg_status( );

**Parameters:**
None

**Returns:**
the status of the CWG module

**Function:**
To determine if a shutdown event occurred causing the module to auto-shutdown

**Availability:**
On devices with a CWG module.
### Examples:
```c
if(cwg_status( ) == CWG_AUTO_SHUTDOWN)
    cwg_restart( );
```

### Example Files:
None

### Also See:
`setup_cwg()`, `cwg_restart()`

---

## cwg_restart()

### Syntax:
```c
cwg_restart( );
```

### Parameters:
None

### Returns:
Nothing

### Function:
To restart the CWG module after an auto-shutdown event occurs, when not using auto-raster option of module.

### Availability:
On devices with a CWG module.

### Examples:
```c
if(cwg_status( ) == CWG_AUTO_SHUTDOWN)
    cwg_restart( );
```

### Example Files:
None

### Also See:
`setup_cwg()`, `cwg_status()`

---

## dac_write()

### Syntax:
```c
dac_write (value)
dac_write (channel, value)
```

### Parameters:
- **Value**: 8-bit integer value to be written to the DAC module
- **Value**: 16-bit integer value to be written to the DAC module
- **Channel**: Channel to be written to. Constants are:
  - `DAC_RIGHT`
Built-in Functions

| DAC_DEFAULT
| DAC_LEFT |

**Returns:** undefined

**Function:** This function will write a 8-bit integer to the specified DAC channel. This function will write a 16-bit integer to the specified DAC channel.

**Availability:** Only available on devices with built in digital to analog converters.

**Requires:** Nothing

**Examples:**

```c
int i = 0;
setup_dac(DAC_VDD | DAC_OUTPUT);
while(1){
  i++;
  dac_write(i);
}
int i = 0;
setup_dac(DAC_RIGHT_ON, 5);
while(1){
  i++;
  dac_write(DAC_RIGHT | i);
}
```

**Also See:** setup_dac(), DAC Overview, see header file for device selected

---

### dci_data_received()

| Syntax: | dci_data_received() |
| Parameters: | none |
| Returns: | An int1. Returns true if the DCI module has received data. |
| Function: | Use this function to poll the receive buffers. It acts as a kbhit() function for DCI. |
| Availability: | Only available on devices with DCI |
| Requires: | None |
Examples:

while(1) {
    if(dci_data_received()) {
        //read data, load buffers, etc…
    }
}

Example Files: None

Also See: DCI Overview, setup_dci(), dci_start(), dci_write(), dci_read(), dci_transmit_ready()

**dci_read( )**

Syntax:      dci_read(left_channel, right_channel);

Parameters: left_channel- A pointer to a signed int16 that will hold the incoming audio data for the left channel (on a stereo system). This data is received on the bus before the right channel data (for situations where left & right channel does have meaning)

right_channel- A pointer to a signed int16 that will hold the incoming audio data for the right channel (on a stereo system). This data is received on the bus after the data in left channel.

Returns: undefined

Function: Use this function to read two data words. Do not use this function with DMA. This function is provided mainly for applications involving a stereo codec.

If your application does not use both channels but only receives on a slot (see setup_dci), use only the left channel.

Availability: Only available on devices with DCI

Requires: None

Examples:

while(1) {
    dci_read(&left_channel, &right_channel);
    dci_write(&left_channel, &right_channel);
}
## dci_start()

**Syntax:**
```c
void dci_start();
```

**Parameters:** None

**Returns:** undefined

**Function:** Starts the DCI module's transmission. DCI operates in a continuous transmission mode (unlike other transmission protocols that transmit only when they have data). This function starts the transmission. This function is primarily provided to use DCI in conjunction with DMA.

**Availability:** Only available on devices with DCI.

**Requires:** None

**Examples:**
```c
  dci_initialize(I2S_MODE | DCI_MASTER | DCI_CLOCK_OUTPUT | SAMPLE_RISING_EDGE | UNDERFLOW_LAST | MULTI_DEVICE_BUS, DCI_1WORD_FRAME | DCI_16BIT_WORD | DCI_2WORD_INTERRUPT, RECEIVE_SLOT0 | RECEIVE_SLOT1, TRANSMIT_SLOT0 | TRANSMIT_SLOT1, 6000);

  ...

  dci_start();
```

**Example Files:** None

**Also See:** DCI Overview, setup_dci(), dci_start(), dci_write(), dci_transmit_ready(), dci_data_received()
**dci_transmit_ready( )**

**Syntax:**

```c
dci_transmit_ready()
```

**Parameters:**

None

**Returns:**

An int1. Returns true if the DCI module is ready to transmit (there is space open in the hardware buffer).

**Function:**

Use this function to poll the transmit buffers.

**Availability:**

Only available on devices with DCI

**Requires:**

None

**Examples:**

```c
while(1)
{
  if(dci_transmit_ready())
  { //transmit data, load buffers, etc…
  }
}
```

**Example Files:**

None

**Also See:**

DCI Overview, setup_dci(), dci_start(), dci_write(), dci_read(), dci_data_received()

---

**dci_write( )**

**Syntax:**

```c
dci_write(left_channel, right_channel);
```

**Parameters:**

- **left channel**: A pointer to a signed int16 that holds the outgoing audio data for the left channel (on a stereo system). This data is transmitted on the bus before the right channel data (for situations where left & right channel does have meaning)

- **right channel**: A pointer to a signed int16 that holds the outgoing audio data for the right channel (on a stereo system). This data is transmitted on the bus after the data in *left channel*.

**Returns:**

undefined
**Function:** Use this function to transmit two data words. Do not use this function with DMA. This function is provided mainly for applications involving a stereo codec.

If your application does not use both channels but only transmits on a slot (see setup_dci()), use only the left channel. If you transmit more than two slots, call this function multiple times.

**Availability:** Only available on devices with DCI

**Requires:** None

**Examples:**
```c
while(1) {
    dci_read(&left_channel, &right_channel);
    dci_write(&left_channel, &right_channel);
}
```

**Example Files:** None

**Also See:** DCI Overview, setup_dci(), dci_start(), dci_read(), dci_transmit_ready(), dci_data_received()

---

**delay_cycles( )**

**Syntax:** `delay_cycles(count)`

**Parameters:**
- `count` - a constant 1-255

**Returns:** undefined

**Function:** Creates code to perform a delay of the specified number of instruction clocks (1-255). An instruction clock is equal to four oscillator clocks.

The delay time may be longer than requested if an interrupt is serviced during the delay. The time spent in the ISR does not count toward the delay time.

**Availability:** All devices

**Requires:** Nothing

**Examples:**
```c
delay_cycles( 1 ); // Same as a NOP
```
delay_cycles(25); // At 20 mhz a 5us delay

Example Files:  ex_cust.c

Also See:  delay_us(), delay_ms()

---

delay_ms()

Syntax:  

```
delay_ms (time)
```

Parameters:  

- `time` - a variable 0-65535(int16) or a constant 0-65535  
  
  Note: Previous compiler versions ignored the upper byte of an int16, now the upper byte affects the time.

Returns:  undefined

Function:  This function will create code to perform a delay of the specified length. Time is specified in milliseconds. This function works by executing a precise number of instructions to cause the requested delay. It does not use any timers. If interrupts are enabled the time spent in an interrupt routine is not counted toward the time.

The delay time may be longer than requested if an interrupt is serviced during the delay. The time spent in the ISR does not count toward the delay time.

Availability:  All devices

Requires:  #USE DELAY

Examples:  

```
#define clock 20000000

void delay_seconds(int n) {
    for (; n!=0; n--) {
        delay_ms( 1000 );
    }
}
```

Example Files:  ex_sqw.c
**delay_us()**

**Syntax:**

```
delay_us (time)
```

**Parameters:**

- `time` - a variable 0-65535(int16) or a constant 0-65535

Note: Previous compiler versions ignored the upper byte of an int16, now the upper byte affects the time.

**Returns:**

undefined

**Function:**

Creates code to perform a delay of the specified length. Time is specified in microseconds. Shorter delays will be INLINE code and longer delays and variable delays are calls to a function. This function works by executing a precise number of instructions to cause the requested delay. It does not use any timers. If interrupts are enabled the time spent in an interrupt routine is not counted toward the time.

The delay time may be longer than requested if an interrupt is serviced during the delay. The time spent in the ISR does not count toward the delay time.

**Availability:**

All devices

**Requires:**

#USE DELAY

**Examples:**

```c
#use delay(clock=20000000)

do {
    output_high(PIN_B0);
    delay_us(duty);
    output_low(PIN_B0);
    delay_us(period-duty);
} while(TRUE);
```

**Example Files:**

ex_sqw.c

**Also See:**

delay_ms(), delay_cycles(), #USE DELAY
disable_interrupts( )

Syntax:

- `disable_interrupts(name)`
- `disable_interrupts(INTR_XX)`
- `disable_interrupts(expression)`

Parameters:

- `name` - a constant defined in the devices .h file
- `INTR_XX` – Allows user selectable interrupt options like INTR_NORMAL, INTR_ALTERNATE, INTR_LEVEL
- `expression` – A non-constant expression

Returns:

- When INTR_LEVELX is used as a parameter, this function will return the previous level.

Function:

Disables the interrupt for the given name. Valid specific names are the same as are used in #INT_xxx and are listed in the devices .h file. Note that it is not necessary to disable interrupts inside an interrupt service routine since interrupts are automatically disabled.

- INTR_GLOBAL – Disables all interrupts that can be disabled
- INTR_NORMAL – Use normal vectors for the ISR
- INTR_ALTERNATE – Use alternate vectors for the ISR
- INTR_LEVEL0 .. INTR_LEVEL7 – Disables interrupts at this level and below, enables interrupts above this level
- INTR_CN_PIN | PIN_xx – Disables a CN pin interrupts
- expression – Disables interrupts during evaluation of the expression.

Availability:

- All dsPIC and PIC24 devices

Requires:

- Should have a #INT_xxxx, constants are defined in the devices .h file.

Examples:

- `disable_interrupts(INT_RDA);` // RS232 OFF
- `disable_interrupts(memcpy(buffer1,buffer2,10));`
- `enable_interrupts(ADC_DONE);`
- `enable_interrupts(RB_CHANGE);`
  // these enable the interrupts
Example Files: None

Also See: enable_interrupts(), #INT xxxx, Interrupts Overview, clear_interrupt(), interrupt_active()

defdisable_pwm1_interrupt()
defdisable_pwm2_interrupt()
defdisable_pwm3_interrupt()
defdisable_pwm4_interrupt()
defdisable_pwm5_interrupt()
defdisable_pwm6_interrupt()

Syntax: disable_pwm1_interrupt (interrupt)
defdisable_pwm2_interrupt (interrupt)
defdisable_pwm3_interrupt (interrupt)
defdisable_pwm4_interrupt (interrupt)
defdisable_pwm5_interrupt (interrupt)
defdisable_pwm6_interrupt (interrupt)

Parameters: interrupt - 8-bit constant or variable. Constants are defined in the device's header file as:
  - PWM_PERIOD_INTERRUPT
  - PWM_DUTY_INTERRUPT
  - PWM_PHASE_INTERRUPT
  - PWM_OFFSET_INTERRUPT

Returns: undefined.

Function: Disables one of the above PWM interrupts, multiple interrupts can be disabled by or'ing multiple options together.

Availability: Devices with a 16-bit PWM module.

Requires: Nothing
Examples:

```c
disable_pwm1_interrupt(PWM_PERIOD_INTERRUPT);
disable_pwm1_interrupt(PWM_PERIOD_INTERRUPT | PWM_DUTY_INTERRUPT);
```

Example Files:

Files:
- `setup_pwm()`
- `set_pwm_duty()`
- `set_pwm_phase()`
- `set_pwm_period()`
- `set_pwm_offset()`
- `enable_pwm_interrupt()`
- `clear_pwm_interrupt()`
- `pwm_interrupt_active()`

### `div( )`  `ldiv( )`

**Syntax:**

```c
idiv = div(num, denom)  
ldiv = ldiv(lnum, ldenom)
```

**Parameters:**
- `num` and `denom` are signed integers.
- `num` is the numerator and `denom` is the denominator.
- `lnum` and `ldenom` are signed longs, signed int32, int48 or int64.
- `lnum` is the numerator and `ldenom` is the denominator.

**Returns:**
- `idiv` is a structure of type `div_t` and `ldiv` is a structure of type `ldiv_t`. The `div` function returns a structure of type `div_t`, comprising of both the quotient and the remainder. The `ldiv` function returns a structure of type `ldiv_t`, comprising of both the quotient and the remainder.

**Function:**
- The `div` and `ldiv` function computes the quotient and remainder of the division of the numerator by the denominator. If the division is inexact, the resulting quotient is the integer or long of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise `quot*denom(ldenom)+rem` shall equal `num(lnum)`.

**Availability:**
- All devices.

**Requires:**
- `#include <stdlib.h>`

**Examples:**

```c
div_t idiv;
ldiv_t ldiv;
idiv = div(3, 2);
// idiv will contain quot=1 and rem=1

ldiv = ldiv(300, 250);
// ldiv will contain ldiv.quot=1 and ldiv.rem=50
```
dma_start( )

Syntax: `dma_start(channel, mode, addressA, addressB, count);`

Parameters:
- **Channel**: The channel used in the DMA transfer
- **mode**: The mode used for the DMA transfer.
- **addressA**: The start RAM address of the buffer to use located within the DMA RAM bank.
- **addressB**: If using PING_PONG mode the start RAM address of the second buffer to use located within the DMA RAM bank.
- **count**: Number of DMA transfers to do. Value must be one less than actual number of transfers.

Returns: `void`

Function: Starts the DMA transfer for the specified channel in the specified mode of operation.

Availability: Devices that have the DMA module.

Requires: Nothing

Examples:
```
dma_start(2, DMA_CONTINOUS | DMA_PING_PONG, 0x4000, 0x4200, 255);
// This will setup the DMA channel 2 for continuous ping-pong mode with DMA RAM addresses of 0x4000 and 0x4200.
```

Example Files: None

Also See: `setup_dma()`, `dma_status()`
### dma_status()

**Syntax:**

\[ \text{Value} = \text{dma_status}(\text{channel}); \]

**Parameters:**

- **Channel** – The channel whose status is to be queried.

**Returns:**

Returns a 8-bit int. Possible return values are:

- DMA_IN_ERROR 0x01
- DMA_OUT_ERROR 0x02
- DMA_B_SELECT 0x04

**Function:**

This function will return the status of the specified channel in the DMA module.

**Availability:**

Devices that have the DMA module.

**Requires:**

Nothing

**Examples:**

```
int8 value;
value = dma_status(3); // This will return the status of channel 3 of the DMA module.
```

**Example Files:**

None

**Also See:**

setup_dma(), dma_start().

### enable_interrupts()

**Syntax:**

```
enable_interrupts (name)
enable_interrupts (INTR_XX)
```

**Parameters:**

- **name** - a constant defined in the devices .h file
- **INTR_XX** – Allows user selectable interrupt options like INTR_NORMAL, INTR_ALTERNATE, INTR_LEVEL

**Returns:**

undefined

**Function:**

**Name** - Enables the interrupt for the given name. Valid specific names are the same as are used in #INT_xxx and are listed in the devices .h file.
**Built-in Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTR_GLOBAL</td>
<td>Enables all interrupt levels (same as INTR_LEVEL0)</td>
</tr>
<tr>
<td>INTR_NORMAL</td>
<td>Use normal vectors for the ISR</td>
</tr>
<tr>
<td>INTR_ALTERNATE</td>
<td>Use alternate vectors for the ISR</td>
</tr>
<tr>
<td>INTR_LEVEL0 .. INTR_LEVEL7</td>
<td>Enables interrupts at this level and above, interrupts at lower levels are disabled</td>
</tr>
<tr>
<td>INTR_CN_PIN</td>
<td>Enables a CN pin interrupts</td>
</tr>
</tbody>
</table>

**Availability:** All dsPIC and PIC24 devices

**Requires:** Should have a #INT_xxxx, Constants are defined in the devices .h file.

**Examples:**

```c
enable_interrupts(INT_TIMER0);
enable_interrupts(INT_TIMER1);
enable_interrupts(INTR_CN_PIN|Pin_B0);
```

**Example Files:** None

**Also See:**

- disable_interrupts()
- #INT_xxxx
- Interrupts Overview
- clear_interrupt()
- interrupt_active()

---

**erase_program_memory**

**Syntax:**

```c
erase_program_memory (address);
```

**Parameters:**

- `address` is 32 bits. The least significant bits may be ignored.

**Returns:** undefined

**Function:**

Erases FLASH_ERASE_SIZE bytes to 0xFFFF in program memory.

- Family
- dsPIC30F 32 instructions (96 bytes)
- dsPIC33FJ 512 instructions (1536 bytes)
- PIC24FJ 512 instructions (1536 bytes)
- PIC24HJ 512 instructions (1536 bytes)

**NOTE:** Each instruction on the PCD is 24 bits wide (3 bytes)

See write_program_memory() for more information on program memory
access.

Availability: All devices

Requires: Nothing

Examples: Int32 address = 0x2000;

erase_program_memory(address); // erase block of memory from 0x2000 to 0x2400 for a PIC24HJ/FJ /33FJ device, or erase 0x2000 to 0x2040 for a dsPIC30F chip

Example Files: None

Also See: write program memory(), Program Eeprom Overview

enable_pwm1_interrupt( )
enable_pwm2_interrupt( )
enable_pwm3_interrupt( )
enable_pwm4_interrupt( )
enable_pwm5_interrupt( )
enable_pwm6_interrupt( )

Syntax: enable_pwm1_interrupt (interrupt)
enable_pwm2_interrupt (interrupt)
enable_pwm3_interrupt (interrupt)
enable_pwm4_interrupt (interrupt)
enable_pwm5_interrupt (interrupt)
enable_pwm6_interrupt (interrupt)

Parameters: interrupt - 8-bit constant or variable. Constants are defined in the device's header file as:
- PWM_PERIOD_INTERRUPT
- PWM_DUTY_INTERRUPT
- PWM_PHASE_INTERRUPT
- PWM_OFFSET_INTERRUPT
## Built-in Functions

<table>
<thead>
<tr>
<th>Returns:</th>
<th>undefined.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function:</strong> Enables one of the above PWM interrupts, multiple interrupts can be enabled by or'ing multiple options together. For the interrupt to occur, the overall PWMx interrupt still needs to be enabled and an interrupt service routine still needs to be created.</td>
<td></td>
</tr>
<tr>
<td><strong>Availability:</strong> Devices with a 16-bit PWM module.</td>
<td></td>
</tr>
<tr>
<td><strong>Requires:</strong> Nothing</td>
<td></td>
</tr>
<tr>
<td><strong>Examples:</strong> enable_pwm1_interrupt(PWM_PERIOD_INTERRUPT); enable_pwm1_interrupt(PWM_PERIOD_INTERRUPT</td>
<td>PWM_DUTY_INTERRUPT);</td>
</tr>
<tr>
<td><strong>Example Files:</strong> Also See:</td>
<td>setup_pwm(), set_pwm_duty(), set_pwm_phase(), set_pwm_period(), set_pwm_offset(), disable_pwm_interrupt(), clear_pwm_interrupt(), pwm_interrupt_active()</td>
</tr>
</tbody>
</table>

### `exp( )`

**Syntax:** `result = exp (value)`

**Parameters:** `value` is any float type

**Returns:** A float with a precision equal to `value`

**Function:** Computes the exponential function of the argument. This is e to the power of value where e is the base of natural logarithms. exp(1) is 2.7182818.

Note on error handling:
If "errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Range error occur in the following case:
- exp: when the argument is too large

**Availability:** All devices
**Requires:**  
```c
#include <math.h>
```

**Examples:**  
```c
// Calculate x to the power of y
x_power_y = exp( y * log(x) );
```

**Example Files:**  
None

**Also See:**  
`pow()`, `log()`, `log10()`

---

**ext_int_edge()**

**Syntax:**  
```c
ext_int_edge(source, edge)
```

**Parameters:**
- `source` is a constant 0, 1 or 2 for the PIC18XXX and 0 otherwise.
- `source` is a constant from 0 to 4.
- Source is optional and defaults to 0.
- `edge` is a constant H_TO_L or L_TO_H representing "high to low" and "low to high"

**Returns:**  
undefined

**Function:**  
Determines when the external interrupt is acted upon. The edge may be L_TO_H or H_TO_L to specify the rising or falling edge.

**Availability:**  
Only devices with interrupts

**Requires:**  
Constants are in the devices .h file

**Examples:**
```c
ext_int_edge( 2, L_TO_H); // Set up PIC18 EXT2
ext_int_edge( 2, L_TO_H); // Set up external interrupt 2 to interrupt
                         // on rising edge
ext_int_edge( H_TO_L ); // Sets up EXT
ext_int_edge( H_TO_L ); // Sets up external interrupt 0 to interrupt
                         // on falling edge
```

**Example Files:**  
ex_wakup.c

**Also See:**  
`#INT_EXT`, `enable_interrupts()`, `disable_interrupts`, `Interrupts Overview`
**fabs( )**

**Syntax:**
```
result=fabs (value)
```

**Parameters:**
`value` is any float type

**Returns:**
result is a float with precision to `value`

**Function:**
The `fabs` function computes the absolute value of a float

**Availability:**
All devices.

**Requires:**
`#INCLUDE <math.h>`

**Examples:**
```
double result;
result=fabs(-40.0)
// result is 40.0
```

**Example Files:**
None

**Also See:**
`abs()`, `labs()`

---

**getc( )**  **getch( )**  **getchar( )**  **fgetc( )**

**Syntax:**
```
value = getc()
value = fgetc(stream)
value=getch()
value=getchar()
```

**Parameters:**
`stream` is a stream identifier (a constant byte)

**Returns:**
An 8 bit character

**Function:**
This function waits for a character to come in over the RS232 RCV pin and returns the character. If you do not want to hang forever waiting for an incoming character use `kbhit()` to test for a character available. If a built-in USART is used the hardware can buffer 3 characters otherwise GETC must be active while the character is being received by the PIC®.
If fgetc() is used then the specified stream is used where getc() defaults to STDIN (the last USE RS232).

**Availability:** All devices

**Requires:** #USE RS232

**Examples:**
```c
printf("Continue (Y,N)?");
do {
    answer=getch();
} while(answer!='Y' && answer!='N');

#use rs232(baud=9600,xmit=pin_c6,
    rcv=pin_c7,stream=HOSTPC)
#use rs232(baud=1200,xmit=pin_b1,
    rcv=pin_b0,stream=GPS)
#use rs232(baud=9600,xmit=pin_b3,
    stream=DEBUG)
...
while(TRUE) {
    c=fgetc(GPS);
    fputc(c,HOSTPC);
    if(c==13)
        fprintf(DEBUG,"Got a CR\r\n");
}
```

**Example Files:** ex_stwt.c

**Also See:** putc(), kbhit(), printf(), #USE RS232, input.c, RS232 I/O Overview

---

**gets( )**

**fgets( )**

**Syntax:**
```c
gets (string)

value = fgets (string, stream)
```

**Parameters:**
- `string` is a pointer to an array of characters.
- `Stream` is a stream identifier (a constant byte)

**Returns:** undefined

**Function:** Reads characters (using getc()) into the string until a RETURN (value 13) is encountered. The string is terminated with a 0. Note that INPUT.C has a more versatile get_string function.
If fgets() is used then the specified stream is used where gets() defaults to STDIN (the last USE RS232).

**Availability:** All devices

**Requires:** #USE RS232

**Examples:**

```c
char string[30];

printf("Password: ");
gets(string);
if(strcmp(string, password))
    printf("OK");
```

**Example Files:** None

**Also See:** getc(), get_string in input.c

### floor( )

**Syntax:**

```c
result = floor (value)
```

**Parameters:**

`value` is any float type

**Returns:**

result is a float with precision equal to `value`

**Function:**

Computes the greatest integer value not greater than the argument. Floor (12.67) is 12.00.

**Availability:** All devices.

**Requires:** #INCLUDE <math.h>

**Examples:**

```c
// Find the fractional part of a value

frac = value - floor(value);
```

**Example Files:** None

**Also See:** ceil()
fmod( )

Syntax: \[ \text{result} = \text{fmod} \left( v_{a1}, v_{a2} \right) \]

Parameters: 
- \( v_{a1} \) is any float type
- \( v_{a2} \) is any float type

Returns: result is a float with precision equal to input parameters \( v_{a1} \) and \( v_{a2} \)

Function: Returns the floating point remainder of \( v_{a1} / v_{a2} \). Returns the value \( v_{a1} - i \cdot v_{a2} \) for some integer “\( i \)” such that, if \( v_{a2} \) is nonzero, the result has the same sign as \( v_{a1} \) and magnitude less than the magnitude of \( v_{a2} \).

Availability: All devices.

Requires: #INCLUDE <math.h>

Examples:
```
float result;
result = fmod(3, 2);
// result is 1
```

Example Files: None

Also See: None

printf( ) fprintf( )

Syntax: 
- printf \( (\text{string}) \)
- or
- printf \( (\text{cstring}, \text{values...}) \)
- or
- printf \( (\text{fname, cstring}, \text{values...}) \)
- or
- fprintf \( (\text{stream, cstring}, \text{values...}) \)

Parameters: 
- \( \text{String} \) is a constant string or an array of characters null terminated.
- \( \text{C String} \) is a constant string. Note that format specifiers cannot be used in RAM strings.
**Values** is a list of variables separated by commas, fname is a function name to be used for outputting (default is putc is none is specified.

**Stream** is a stream identifier (a constant byte).

**Returns:** undefined

**Function:** Outputs a string of characters to either the standard RS-232 pins (first two forms) or to a specified function. Formatting is in accordance with the string argument. When variables are used this string must be a constant. The % character is used within the string to indicate a variable value is to be formatted and output. Longs in the printf may be 16 or 32 bit. A % will output a single %. Formatting rules for the % follows.

See the Expressions > Constants and Trigraph sections of this manual for other escape character that may be part of the string.

If fprintf() is used then the specified stream is used where printf() defaults to STDOUT (the last USE RS232).

**Format:**
The format takes the generic form %nt. n is optional and may be 1-9 to specify how many characters are to be outputted, or 01-09 to indicate leading zeros, or 1.1 to 9.9 for floating point and %w output. t is the type and may be one of the following:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Value=0x12</th>
<th>Value=0xfe</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Character</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>String or character</td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>Unsigned int</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Signed int</td>
<td></td>
</tr>
<tr>
<td>Lu</td>
<td>Long unsigned int</td>
<td></td>
</tr>
<tr>
<td>Ld</td>
<td>Long signed int</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>Hex int (lower case)</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>Hex int (upper case)</td>
<td></td>
</tr>
<tr>
<td>Lx</td>
<td>Hex long int (lower case)</td>
<td></td>
</tr>
<tr>
<td>LX</td>
<td>Hex long int (upper case)</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>Float with truncated decimal</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>Float with rounded decimal</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>Float in exponential format</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>Unsigned int with decimal place inserted. Specify two numbers for n. The first is a total field width. The second is the desired number of decimal places.</td>
<td></td>
</tr>
</tbody>
</table>
**Availability:** All Devices

**Requires:** #USE RS232 (unless fname is used)

**Examples:**
```
byte x,y,z;
printf("HiThere\n");
printf("RTCCValue=%2x\n",get_rtcc());
printf("%2u %X %4X\n",x,y,z);
printf(LCD_PUTC, "n=%u",n);
```

**Example Files:** ex_admm.c, ex_lcdkb.c

**Also See:** atoi(), puts(), putc(), getc() (for a stream example), RS232 I/O Overview

---

**putc( ) putchar( ) fputc( )**

**Syntax:**
- `putc(cdata)`
- `putchar(cdata)`
- `fputc(cdata, stream)`

**Parameters:**
- `cdata` is a 8 bit character.
- `Stream` is a stream identifier (a constant byte)

**Returns:** undefined

**Function:** This function sends a character over the RS232 XMIT pin. A #USE RS232 must appear before this call to determine the baud rate and pin used. The
#USE RS232 remains in effect until another is encountered in the file.

If `fputc()` is used then the specified stream is used where `putc()` defaults to
Built-in Functions

**STDOUT** (the last USE RS232).

**Availability:** All devices

**Requires:** #USE RS232

**Examples:**
```c
putc('*');
for(i=0; i<10; i++)
    putc(buffer[i]);
putc(13);
```

**Example Files:** ex_tgetc.c

**Also See:** getc(), printf(), #USE RS232, RS232 I/O Overview

---

**puts( ) fputs( )**

**Syntax:**
- `puts (string).
- `fputs (string, stream)`

**Parameters:**
- `string` is a constant string or a character array (null-terminated).
- `Stream` is a stream identifier (a constant byte)

**Returns:** undefined

**Function:** Sends each character in the string out the RS232 pin using putc(). After the string is sent a CARRIAGE-RETURN (13) and LINE-FEED (10) are sent. In general printf() is more useful than puts().

If fputs() is used then the specified stream is used where puts() defaults to STDOUT (the last USE RS232)

**Availability:** All devices

**Requires:** #USE RS232

**Examples:**
```c
puts( " -------------- ");
puts( " | HI | ");
puts( " -------------- ");
```

**Example Files:** None

**Also See:** printf(), gets(), RS232 I/O Overview
### free( )

**Syntax:**
\[
\text{free}(\text{ptr})
\]

**Parameters:**
\[
\text{ptr} \text{ is a pointer earlier returned by the calloc, malloc or realloc.}
\]

**Returns:**
No value

**Function:**
The free function causes the space pointed to by the ptr to be deallocated, that is made available for further allocation. If ptr is a null pointer, no action occurs. If the ptr does not match a pointer earlier returned by the calloc, malloc or realloc, or if the space has been deallocated by a call to free or realloc function, the behavior is undefined.

**Availability:**
All devices.

**Requires:**
#include <stdlib.h>

**Examples:**
```c
int * iptr;
iptr=malloc(10);
free(iptr)
// iptr will be deallocated
```

**Example Files:**
None

**Also See:**
realloc(), malloc(), calloc()

### frexp( )

**Syntax:**
\[
\text{result=}\text{frexp}(\text{value}, \&\text{exp});
\]

**Parameters:**
\[
\text{value} \text{ is any float type}
\]
\[
\text{exp} \text{ is a signed int.}
\]

**Returns:**
result is a float with precision equal to \text{value}

**Function:**
The frexp function breaks a floating point number into a normalized fraction and an integral power of 2. It stores the integer in the signed int object exp. The result is in the interval [1/2 to1) or zero, such that value is result times 2 raised to power exp. If value is zero then both parts are zero.
Built-in Functions

Availability: All devices.

Requires: #include <math.h>

Examples:
float result;
signed int exp;
result=frexp(.5,&exp);
// result is .5 and exp is 0

Example Files: None

Also See: ldexp(), exp(), log(), log10(), modf()

scanf()

Syntax:
scanf(cstring);
scanf(cstring, values...)
fscanf(stream, cstring, values...)

Parameters:
cstring is a constant string.
values is a list of variables separated by commas.
stream is a stream identifier.

Returns: 0 if a failure occurred, otherwise it returns the number of conversion specifiers that were read in, plus the number of constant strings read in.

Function: Reads in a string of characters from the standard RS-232 pins and formats the string according to the format specifiers. The format specifier character (%) used within the string indicates that a conversion specification is to be done and the value is to be saved into the corresponding argument variable. A % will input a single %. Formatting rules for the format specifier as follows:

If fscanf() is used, then the specified stream is used, where scanf() defaults to STDIN (the last USE RS232).

Format:
The format takes the generic form %nt. n is an option and may be 1-99 specifying the field width, the number of characters to be inputted. t is the
type and maybe one of the following:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Corresponding Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Matches a sequence of characters of the number specified by the field width (1 if no field width is specified). The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence.</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>Matches a sequence of non-white space characters. The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence and a terminating null character, which will be added automatically.</td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>Matches an unsigned decimal integer. The corresponding argument shall be a pointer to an unsigned integer.</td>
<td></td>
</tr>
<tr>
<td>Lu</td>
<td>Matches a long unsigned decimal integer. The corresponding argument shall be a pointer to a long unsigned integer.</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Matches a signed decimal integer. The corresponding argument shall be a pointer to a signed integer.</td>
<td></td>
</tr>
<tr>
<td>Ld</td>
<td>Matches a long signed decimal integer. The corresponding argument shall be a pointer to a long signed integer.</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>Matches a signed or unsigned octal integer. The corresponding argument shall be a pointer to a signed or unsigned integer.</td>
<td></td>
</tr>
<tr>
<td>Lo</td>
<td>Matches a long signed or unsigned octal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.</td>
<td></td>
</tr>
<tr>
<td>x or X</td>
<td>Matches a hexadecimal integer. The corresponding argument shall be a pointer to a signed or unsigned integer.</td>
<td></td>
</tr>
<tr>
<td>Lx or LX</td>
<td>Matches a long hexadecimal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>Matches a signed or unsigned integer. The corresponding argument shall be a pointer to a signed or unsigned integer.</td>
<td></td>
</tr>
</tbody>
</table>
**Built-in Functions**

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Li</strong></td>
<td>Matches a long signed or unsigned integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.</td>
</tr>
<tr>
<td><strong>f,g or e</strong></td>
<td>Matches a floating point number in decimal or exponential format. The corresponding argument shall be a pointer to a float.</td>
</tr>
<tr>
<td>[</td>
<td>Matches a non-empty sequence of characters from a set of expected characters. The sequence of characters included in the set are made up of all character following the left bracket ([) up to the matching right bracket (]). Unless the first character after the left bracket is a ^, in which case the set of characters contain all characters that do not appear between the brackets. If a - character is in the set and is not the first or second, where the first is a ^, nor the last character, then the set includes all characters from the character before the - to the character after the -. For example, %[a-z] would include all characters from a to z in the set and %[^a-z] would exclude all characters from a to z from the set. The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence and a terminating null character, which will be added automatically.</td>
</tr>
<tr>
<td><strong>n</strong></td>
<td>Assigns the number of characters read thus far by the call to scanf() to the corresponding argument. The corresponding argument shall be a pointer to an unsigned integer. An optional assignment-suppressing character (*) can be used after the format specifier to indicate that the conversion specification is to be done, but not saved into a corresponding variable. In this case, no corresponding argument variable should be passed to the scanf() function.</td>
</tr>
</tbody>
</table>

A string composed of ordinary non-white space characters is executed by reading the next character of the string. If one of the inputted characters differs from the string, the function fails and exits. If a white-space character precedes the ordinary non-white space characters, then
white-space characters are first read in until a non-white space character is read.

White-space characters are skipped, except for the conversion specifiers [], c or n, unless a white-space character precedes the [ or c specifiers.

Availability: All Devices

Requires: #USE RS232

Examples:

```c
char name[2-];
unsigned int8 number;
signed int32 time;

if(scanf("%u%s%ld",&number,name,&time))
    printf"\r\nName: %s, Number: %u, Time: %ld",name,number,time);
```

Example Files: None

Also See: RS232 I/O Overview, getc(), putc(), printf()

---

get_capture( )

Syntax: value = get_capture(x)

Parameters: x defines which ccp module to read from.

Returns: A 16-bit timer value.

Function: This function obtains the last capture time from the indicated CCP module

Availability: Only available on devices with Input Capture modules

Requires: None

Examples:

Example Files: ex_ccpmp.c

Also See: setup_ccpx( )
get_capture( )

Syntax: value = get_capture(x, wait)

Parameters: x defines which input capture result buffer module to read from. wait signifies if the compiler should read the oldest result in the buffer or the next result to enter the buffer.

Returns: A 16-bit timer value.

Function: If wait is true, the current capture values in the result buffer are cleared, and the next result to be sent to the buffer is returned. If wait is false, the default setting, the first value currently in the buffer is returned. However, the buffer will only hold four results while waiting for them to be read, so if read isn’t being called for every capture event, when wait is false, the buffer will fill with old capture values and any new results will be lost.

Availability: Only available on devices with Input Capture modules.

Requires: None

Examples: setup_timer3(TMR_INTERNAL | TMR_DIV_BY_8);
setup_capture(2, CAPTURE_FE | CAPTURE_TIMER3);
while(TRUE) {
    timerValue = get_capture(2, TRUE);
    printf("Capture 2 occurred at: %LU", timerValue);
}

Example Files: None
Also See: setup_capture(), setup_compare(), Input Capture Overview
get_capture_ccp1()  get_capture_ccp2()
get_capture_ccp3()  get_capture_ccp4()
get_capture_ccp5()

Syntax: value=get_capture_ccpX(wait);

Parameters: wait - signifies if the compiler should read the oldest result in the buffer or the next result in the buffer or the next result to enter the buffer.

Returns: value16 - a 16-bit timer value

Function: If wait is true, the current capture values in the result buffer are cleared, and the next result to be sent, the buffer is returned. If wait is false, the default setting, the first value currently in the buffer is return. However, the buffer will only hold four results while waiting for them to be read. If read is not being called for every capture event, when wait is false, the buffer will fill with old capture values and any new result will be lost.

Availability: Available only on PIC24FxxKMxxx family of devices with a MCCP and/or SCCP modules.

Requires: Nothing

Examples: unsigned int16 value;

setup_ccp1(CCP_CAPTURE_FE);

while(TRUE) {
  value=get_capture_ccp1(TRUE);
  printf("Capture occurred at: %LU", value);
}

Example Files: None

Also See: set_pwmX_duty(), setup_ccpX(), set_ccpX_compare_time(), set_timer_ccpX(), set_timer_period_ccpX(), get_timer_ccpx(), get_capture32_ccpX()
### Built-in Functions

**get_capture32_ccp1()  get_capture32_ccp2()**  
**get_capture32_ccp3()  get_capture32_ccp4()**  
**get_capture32_ccp5()**

<table>
<thead>
<tr>
<th>Syntax:</th>
<th>value = get_capture32_ccpX(wait);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters:</td>
<td>wait -signifies if the compiler should read the oldest result in the buffer or the next result in the buffer or the next result to enter the buffer.</td>
</tr>
<tr>
<td>Returns:</td>
<td>value32 -a 32-bit timer value</td>
</tr>
<tr>
<td>Function:</td>
<td>If wait is true, the current capture values in the result buffer are cleared, and the next result to be sent, the buffer is returned. If wait is false, the default setting, the first value currently in the buffer is return. However, the buffer will only hold two results while waiting for them to be read. If read is not being called for every capture event, when wait is false, the buffer will fill with old capture values and any new result will be lost.</td>
</tr>
<tr>
<td>Availability:</td>
<td>Available only on PIC24FxxKMXxx family of devices with a MCCP and/or SCCP modules.</td>
</tr>
<tr>
<td>Requires:</td>
<td>Nothing</td>
</tr>
<tr>
<td>Examples:</td>
<td>unsigned int32 value;</td>
</tr>
<tr>
<td></td>
<td>setup_ccp1(CCP_CAPTURE_FE</td>
</tr>
<tr>
<td></td>
<td>while(TRUE) {</td>
</tr>
<tr>
<td></td>
<td>value = get_capture_ccp1(TRUE);</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;Capture occurred at: %LU&quot;, value);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Example Files:</td>
<td>None</td>
</tr>
<tr>
<td>Also See:</td>
<td>set_pwmX_duty(), setup_ccpX(), set_ccpX_compare_time(), set_timer_ccpX(), set_timer_period_ccpX(), get_timer_ccpX(), get_capture_ccpX()</td>
</tr>
</tbody>
</table>
## get_capture_event()

**Syntax:**
```c
result = get_capture_event([stream]);
```

**Parameters:**
| stream | optional parameter specifying the stream defined in #USE CAPTURE. |

**Returns:**
TRUE if a capture event occurred, FALSE otherwise.

**Function:**
To determine if a capture event occurred.

**Availability:**
All devices.

**Requires:**
#USE CAPTURE

**Examples:**
```c
#USE CAPTURE(INPUT=PIN_C2,CAPTURE_RISING,TIMER=1,FASTEST)
if(get_capture_event())
    result = get_capture_time();
```

**Example Files:**
None

**Also See:**
#use_capture, get_capture_time()

## get_capture_time()

**Syntax:**
```c
result = get_capture_time([stream]);
```

**Parameters:**
| stream | optional parameter specifying the stream defined in #USE CAPTURE. |

**Returns:**
An int16 value representing the last capture time.

**Function:**
To get the last capture time.

**Availability:**
All devices.

**Requires:**
#USE CAPTURE
### Examples:

```c
#USE_CAPTURE(INPUT=PIN_C2,CAPTURE_RISING,TIMER=1,FASTEST)
result = get_capture_time();
```

### Example Files:

None

### Also See:

#use_capture, get_capture_event()

---

### get_capture32()

#### Syntax:

```c
result = get_capture32(x,[wait]);
```

#### Parameters:

- `x` is 1-16 and defines which input capture result buffer modules to read from.
- `wait` is an optional parameter specifying if the compiler should read the oldest result in the buffer or the next result to enter the buffer.

#### Returns:

A 32-bit timer value

#### Function:

If `wait` is true, the current capture values in the result buffer are cleared, and the next result to be sent to the buffer is returned. If `wait` is false, the default setting, the first value currently in the buffer is returned. However, the buffer will only hold four results while waiting for them to be read, so if get_capture32 is not being called for every capture event. When `wait` is false, the buffer will fill with old capture values and any new results will be lost.

#### Availability:

Only devices with a 32-bit Input Capture module

#### Requires:

Nothing

#### Examples:

```c
setup_timer2(TMR_INTERNAL | TMR_DIV_BY_1 | TMR_32_BIT);
setup_capture(1,CAPTURE_FE | CAPTURE_TIMER2 | CAPTURE_32_BIT);
while(TRUE) {
    timerValue=get_capture32(1,TRUE);
    printf("Capture 1 occurred at: %LU", timerValue);
}
```

#### Example Files:

None

#### Also See:

setup_capture(), setup_compare(), get_capture(), Input Capture Overview
### get_hspwm_capture( )

**Syntax:**
```
result = get_hspwm_capture(unit);
```

**Parameters:**
- `unit` - The High Speed PWM unit to set.

**Returns:**
Unsigned in16 value representing the capture PWM time base value.

**Function:**
Gets the captured PWM time base value from the leading edge detection on the current-limit input.

**Availability:**

**Requires:**
None

**Examples:**
```
result = get_hspwm_capture(1);
```

**Example Files:**
None

**Also See:**
- `setup_hspwm_unit()`, `set_hspwm_phase()`, `set_hspwm_duty()`, `set_hspwm_event()`, `setup_hspwm_blanking()`, `setup_hspwm_trigger()`, `set_hspwm_override()`, `setup_hspwm_chop_clock()`, `setup_hspwm_unit_chop_clock()`, `setup_hspwm()`, `setup_hspwm_secondary()`

---

### get_motor_pwm_count( )

**Syntax:**
```
Data16 = get_motor_pwm_count(pwm);
```

**Parameters:**
- `pwm` - Defines the pwm module used.

**Returns:**
16 bits of data

**Function:**
Returns the PWM count of the motor control unit.

**Availability:**
Devices that have the motor control PWM unit.

**Requires:**
None
### get_nco_accumulator( )

**Syntax:**

```c
value = get_nco_accumulator();
```

**Parameters:**

None

**Returns:**

Current value of accumulator.

**Availability:**

On devices with a NCO module.

**Examples:**

```c
value = get_nco_accumulator();
```

**Example Files:**

None

**Also See:**

setup_nco(), set_nco_inc_value(), get_nco_inc_value()

---

### get_nco_inc_value( )

**Syntax:**

```c
value = get_nco_inc_value();
```

**Parameters:**

None

**Returns:**

Current value set in increment registers.

**Availability:**

On devices with a NCO module.

**Examples:**

```c
value = get_nco_inc_value();
```

**Example Files:**

None

**Also See:**

setup_nco(), set_nco_inc_value(), get_nco_accumulator()
**get_ticks( )**

Syntax:  
```plaintext
value = get_ticks([stream]);
```

Parameters:  
`stream` – optional parameter specifying the stream defined in `#USE TIMER`.  

Returns:  
`value` – a 8, 16, 32 or 64 bit integer. (int8, int16, int32 or int64)

Function:  
Returns the current tick value of the tick timer. The size returned depends on the size of the tick timer.

Availability:  
All devices.

Requires:  
`#USE TIMER(options)`

Examples:  
```plaintext
#USE TIMER(TIMER=1,TICK=1ms,BITS=16,NOISR)

void main(void) {
    unsigned int16 current_tick;
    current_tick = get_ticks();
}
```

File:  
None

Also See:  
`#USE TIMER, set_ticks()`

---

**get_timerA( )**

Syntax:  
```plaintext
value=get_timerA();
```

Parameters:  
none

Returns:  
The current value of the timer as an int8

Function:  
Returns the current value of the timer. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2, …).

Availability:  
This function is only available on devices with Timer A hardware.

Requires:  
Nothing

Examples:  
```plaintext
set_timerA(0);
```
while(timerA < 200);

Example Files: none
Also See: set_timerA(), setup_timer_A(), TimerA Overview

get_timerB()

Syntax: value=get_timerB();
Parameters: none
Returns: The current value of the timer as an int8
Function: Returns the current value of the timer. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2, …).
Availability: This function is only available on devices with Timer B hardware.
Requires: Nothing
Examples: set_timerB(0);
while(timerB < 200);

Example Files: none
Also See: set_timerB(), setup_timer_B(), TimerB Overview

get_timerx()

Syntax: value=get_timer1();
value=get_timer2();
value=get_timer3();
value=get_timer4();
value=get_timer5();
value=get_timer6();
value=get_timer7();
value=get_timer8();
**get_timer9( )**

**Parameters:** None

**Returns:** The current value of the timer as an int16

**Function:** Retrieves the value of the timer, specified by X (which may be 1-9)

**Availability:** This function is available on all devices that have a valid timerX.

**Requires:** Nothing

**Examples:**

```c
if(get_timer2() % 0xA0 == HALF_WAVE_PERIOD)
    output_toggle(PIN_B0);
```

**Example Files:** `ex_stwt.c`

**Also See:** Timer Overview, setup_timerX(), get_timerXY(), set_timerX(), set_timerXY()

---

**get_timerxy( )**

**Syntax:**

```c
value=get_timer23( )
value=get_timer45( )
value=get_timer67( )
value=get_timer89( )
```

**Parameters:** Void

**Returns:** The current value of the 32 bit timer as an int32

**Function:** Retrieves the 32 bit value of the timers X and Y, specified by XY (which may be 23, 45, 67 and 89)

**Availability:** This function is available on all devices that have a valid 32 bit enabled timers. Timers 2 & 3, 4 & 5, 6 & 7 and 8 & 9 may be used. The target device must have one of these timer sets. The target timers must be enabled as 32 bit.

**Requires:** Nothing

**Examples:**

```c
if(get_timer23() > TRIGGER_TIME)
    ExecuteEvent();
```
get_timer_ccp1() get_timer_ccp2() get_timer_ccp3() get_timer_ccp4() get_timer_ccp5()

Syntax: value32 = get_timer_ccpx();
value16 = get_timer_ccpx(which);

Parameters: which - when in 16-bit mode determines which timer value to read. 0 reads the lower timer value (CCPxTMRL), and 1 reads the upper timer value (CCPxTMRH).

Returns: value32 - the 32-bit timer value.
value16 - the 16-bit timer value.

Function: This function gets the timer values for the CCP module.

Availability: Available only on PIC24FxxKMxxx family of devices with a MCCP and/or SCCP modules.

Requires: Nothing

Examples:
unsigned int32 value32;
unsigned int32 value15;
value32 = get_timer_ccpx(); //get the 32 bit timer value
value16 = get_timer_ccpx(0); //get the 16 bit timer value from
from
value16 = get_timer_ccpx(1); //get the 16 bit timer value
from
//lower timer
//upper timer
**get_tris_x( )**

**Syntax:**
```
value = get_tris_A();
value = get_tris_B();
value = get_tris_C();
value = get_tris_D();
value = get_tris_E();
value = get_tris_F();
value = get_tris_G();
value = get_tris_H();
value = get_tris_J();
value = get_tris_K();
```

**Parameters:** None

**Returns:** int16, the value of TRIS register

**Function:** Returns the value of the TRIS register of port A, B, C, D, E, F, G, H, J, or K.

**Availability:** All devices.

**Requires:** Nothing

**Examples:**
```
tris_a = GET_TRIS_A();
```

**Example Files:** None

**Also See:** input(), output_low(), output_high()
**getenv( )**

**Syntax:**
```c
value = getenv (cstring);
```

**Parameters:**
cstring is a constant string with a recognized keyword

**Returns:**
A constant number, a constant string or 0

**Function:**
This function obtains information about the execution environment. The following are recognized keywords. This function returns a constant 0 if the keyword is not understood.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUSE_SET:fffff</td>
<td>Returns 1 if fuse fffff is enabled</td>
</tr>
<tr>
<td>FUSE_VALID:fffff</td>
<td>Returns 1 if fuse fffff is valid</td>
</tr>
<tr>
<td>INT:iiiii</td>
<td>Returns 1 if the interrupt iiiii is valid</td>
</tr>
<tr>
<td>ID</td>
<td>Returns the device ID (set by #ID)</td>
</tr>
<tr>
<td>DEVICE</td>
<td>Returns the device name string (like &quot;PIC16C74&quot;)</td>
</tr>
<tr>
<td>CLOCK</td>
<td>Returns the MPU FOSC</td>
</tr>
<tr>
<td>VERSION</td>
<td>Returns the compiler version as a float</td>
</tr>
<tr>
<td>VERSION_STRING</td>
<td>Returns the compiler version as a string</td>
</tr>
<tr>
<td>PROGRAM_MEMORY</td>
<td>Returns the size of memory for code (in words)</td>
</tr>
<tr>
<td>STACK</td>
<td>Returns the stack size</td>
</tr>
<tr>
<td>SCRATCH</td>
<td>Returns the start of the compiler scratch area</td>
</tr>
<tr>
<td>DATA_EEPROM</td>
<td>Returns the number of bytes of data EEPROM</td>
</tr>
</tbody>
</table>
| EEPROM_ADDRESS    | Returns the address of the start of EEPROM. 0 if not supported by the device.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_PROGRAM</td>
<td>Returns a 1 if the code memory can be read</td>
</tr>
<tr>
<td>ADC_CHANNELS</td>
<td>Returns the number of A/D channels</td>
</tr>
<tr>
<td>ADC_RESOLUTION</td>
<td>Returns the number of bits returned from READ_ADC()</td>
</tr>
<tr>
<td>ICD</td>
<td>Returns a 1 if this is being compiled for an ICD</td>
</tr>
<tr>
<td>SPI</td>
<td>Returns a 1 if the device has SPI</td>
</tr>
<tr>
<td>USB</td>
<td>Returns a 1 if the device has USB</td>
</tr>
<tr>
<td>CAN</td>
<td>Returns a 1 if the device has CAN</td>
</tr>
<tr>
<td>I2C_SLAVE</td>
<td>Returns a 1 if the device has I2C slave H/W</td>
</tr>
<tr>
<td>I2C_MASTER</td>
<td>Returns a 1 if the device has I2C master H/W</td>
</tr>
<tr>
<td>PSP</td>
<td>Returns a 1 if the device has PSP</td>
</tr>
<tr>
<td>COMP</td>
<td>Returns a 1 if the device has a comparator</td>
</tr>
<tr>
<td>VREF</td>
<td>Returns a 1 if the device has a voltage reference</td>
</tr>
<tr>
<td>LCD</td>
<td>Returns a 1 if the device has direct LCD H/W</td>
</tr>
<tr>
<td>UART</td>
<td>Returns the number of H/W UARTs</td>
</tr>
<tr>
<td>AUART</td>
<td>Returns 1 if the device has an ADV UART</td>
</tr>
<tr>
<td>CCPx</td>
<td>Returns a 1 if the device has CCP number x</td>
</tr>
<tr>
<td>TIMERx</td>
<td>Returns a 1 if the device has TIMER number x</td>
</tr>
<tr>
<td>FLASH_WRITE_SIZE</td>
<td>Smallest number of bytes that can be written to FLASH</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FLASH_ERASE_SIZE</td>
<td>Smallest number of bytes that can be erased in FLASH</td>
</tr>
<tr>
<td>BYTES_PER_ADDRESS</td>
<td>Returns the number of bytes at an address location</td>
</tr>
<tr>
<td>BITS_PER_INSTRUCTION</td>
<td>Returns the size of an instruction in bits</td>
</tr>
<tr>
<td>RAM</td>
<td>Returns the number of RAM bytes available for your device.</td>
</tr>
<tr>
<td>SFR:name</td>
<td>Returns the address of the specified special file register. The output format can be used with the preprocessor command <code>#bit. name</code> must match SFR denomination of your target PIC (example: STATUS, INTCON, TXREG, RCREG, etc)</td>
</tr>
<tr>
<td>BIT:name</td>
<td>Returns the bit address of the specified special file register bit. The output format will be in “address:bit”, which can be used with the preprocessor command <code>#byte. name</code> must match SFR.bit denomination of your target PIC (example: C, Z, GIE, TMR0IF, etc)</td>
</tr>
<tr>
<td>SFR_VALID:name</td>
<td>Returns TRUE if the specified special file register name is valid and exists for your target PIC (example: <code>getenv(&quot;SFR_VALID:INTCON&quot;)</code>)</td>
</tr>
<tr>
<td>BIT_VALID:name</td>
<td>Returns TRUE if the specified special file register bit is valid and exists for your target PIC (example: <code>getenv(&quot;BIT_VALID:TMR0IF&quot;)</code>)</td>
</tr>
<tr>
<td>PIN:PB</td>
<td>Returns 1 if PB is a valid I/O PIN (like A2)</td>
</tr>
<tr>
<td>UARTx_RX</td>
<td>Returns UARTxPin (like PINxC7)</td>
</tr>
<tr>
<td>UARTx_TX</td>
<td>Returns UARTxPin (like PINxC6)</td>
</tr>
<tr>
<td>SPIx_DI</td>
<td>Returns SPIxDI Pin</td>
</tr>
<tr>
<td>SPIxDO</td>
<td>Returns SPIxDO Pin</td>
</tr>
<tr>
<td>SPIxCLK</td>
<td>Returns SPIxCLK Pin</td>
</tr>
<tr>
<td>ETHERNET</td>
<td>Returns 1 if device supports Ethernet</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>QEI</td>
<td>Returns 1 if device has QEI</td>
</tr>
<tr>
<td>DAC</td>
<td>Returns 1 if device has a D/A Converter</td>
</tr>
<tr>
<td>DSP</td>
<td>Returns 1 if device supports DSP instructions</td>
</tr>
<tr>
<td>DCI</td>
<td>Returns 1 if device has a DCI module</td>
</tr>
<tr>
<td>DMA</td>
<td>Returns 1 if device supports DMA</td>
</tr>
<tr>
<td>CRC</td>
<td>Returns 1 if device has a CRC module</td>
</tr>
<tr>
<td>CWG</td>
<td>Returns 1 if device has a CWG module</td>
</tr>
<tr>
<td>NCO</td>
<td>Returns 1 if device has a NCO module</td>
</tr>
<tr>
<td>CLC</td>
<td>Returns 1 if device has a CLC module</td>
</tr>
<tr>
<td>DSM</td>
<td>Returns 1 if device has a DSM module</td>
</tr>
<tr>
<td>OPAMP</td>
<td>Returns 1 if device has op amps</td>
</tr>
<tr>
<td>RTC</td>
<td>Returns 1 if device has a Real Time Clock</td>
</tr>
<tr>
<td>CAP_SENSE</td>
<td>Returns 1 if device has a CSM cap sense module and 2 if it has a CTMU module</td>
</tr>
<tr>
<td>EXTERNAL_MEMORY</td>
<td>Returns 1 if device supports external program memory</td>
</tr>
<tr>
<td>INSTRUCTION_CLOCK</td>
<td>Returns the MPU instruction clock</td>
</tr>
<tr>
<td>ENH16</td>
<td>Returns 1 for Enhanced 16 devices</td>
</tr>
<tr>
<td>ENH24</td>
<td>Returns 2 for Enhanced 24 devices</td>
</tr>
<tr>
<td>IC</td>
<td>Returns number of Input Capture units device has</td>
</tr>
<tr>
<td>ICx</td>
<td>Returns TRUE if ICx is on this part</td>
</tr>
<tr>
<td>OC</td>
<td>Returns number of Output Compare units device has</td>
</tr>
</tbody>
</table>
### Built-in Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OCx</strong></td>
<td>Returns TRUE if OCx is on this part</td>
</tr>
<tr>
<td><strong>RAM_START</strong></td>
<td>Returns the starting address of the first general purpose RAM location</td>
</tr>
<tr>
<td><strong>PSV</strong></td>
<td>Returns TRUE if program space visibility (PSV) is enabled. If PSV is enabled, data in program memory (<code>const char *</code> or <code>rom char *</code>) can be assigned to a regular RAM pointer (<code>char *</code>) and a regular RAM pointer can dereference data from program memory or RAM.</td>
</tr>
</tbody>
</table>

**Availability:** All devices

**Requires:** Nothing

**Examples:**

```c
#if getenv("VERSION")<3.050
  #ERROR  Compiler version too old
#endif

for(i=0;i<getenv("DATA_EEPROM");i++)
  write_eeprom(i,0);
#if getenv("FUSE_VALID:BROWNOUT")
  #FUSE BROWNOUT
#endif

#define status_reg=GETENV("SFR:STATUS")

#define carry_flag=GETENV("BIT:C")
```

**Example Files:** None

**Also See:** None
**goto_address( )**

**Syntax:**

\[ \text{goto_address(location);} \]

**Parameters:**

location is a ROM address, 16 or 32 bit int.

**Returns:**

Nothing

**Function:**

This function jumps to the address specified by location. Jumps outside of the current function should be done only with great caution. This is not a normally used function except in very special situations.

**Availability:**

All devices

**Requires:**

Nothing

**Examples:**

```c
#define LOAD_REQUEST PIN_B1
#define LOADER 0x1f00
if(input(LOAD_REQUEST))
    goto_address(LOADER);
```

**Example Files:**

setjmp.h

**Also See:**

label_address( )

---

**high_speed_adc_done( )**

**Syntax:**

\[ \text{value = high_speed_adc_done([pair]);} \]

**Parameters:**

pair – Optional parameter that determines which ADC pair's ready flag to check. If not used all ready flags are checked.

**Returns:**

An int16. If pair is used 1 will be return if ADC is done with conversion, 0 will be return if still busy. If pair isn't use it will return a bit map of which conversion are ready to be read. For example a return value of 0x0041 means that ADC pair 6, AN12 and AN13, and ADC pair 0, AN0 and AN1, are ready to be read.

**Function:**

Can be polled to determine if the ADC has valid data to be read.
Built-in Functions

**Availability:** Only on dsPIC33FJxxGSxxx devices.

**Requires:** None

**Examples:**

```c
int16 result[2]
setup_high_speed_adc_pair(1, INDIVIDUAL_SOFTWARE_TRIGGER);
setup_high_speed_adc(ADC_CLOCK_DIV_4);

read_high_speed_adc(1, ADC_START_ONLY);
while(!high_speed_adc_done(1));
read_high_speed_adc(1, ADC_READ_ONLY, result);
printf("AN2 value = %LX, AN3 value = %LX\n\n", result[0], result[1]);
```

**Example Files:** None

**Also See:**
- setup_high_speed_adc()
- setup_high_speed_adc_pair()
- read_high_speed_adc()

---

**i2c_init()**

**Syntax:**

```c
i2c_init([stream], baud);
```

**Parameters:**
- `stream` – optional parameter specifying the stream defined in #USE I2C.
- `baud` – if baud is 0, I2C peripheral will be disable. If baud is 1, I2C peripheral is initialized and enabled with baud rate specified in #USE I2C directive. If baud is > 1 then I2C peripheral is initialized and enabled to specified baud rate.

**Returns:** Nothing

**Function:** To initialize I2C peripheral at run time to specified baud rate.

**Availability:** All devices.

**Requires:** #USE I2C

**Examples:**

```c
#USE I2C(MASTER,I2C1, FAST,NOINIT)
i2c_init(TRUE); //initialize and enable I2C peripheral to baud rate specified in //#USE I2C
i2c_init(500000); //initialize and enable I2C peripheral to a baud rate of 500 //KBPS
```
**i2c_isr_state()**

**Syntax:**

```c
state = i2c_isr_state();
state = i2c_isr_state(stream);
```

**Parameters:** None

**Returns:** State is an 8 bit int

- 0 - Address match received with R/W bit clear, perform i2c_read() to read the I2C address.
- 1-0x7F - Master has written data; i2c_read() will immediately return the data
- 0x80 - Address match received with R/W bit set; perform i2c_read() to read the I2C address, and use i2c_write() to pre-load the transmit buffer for the next transaction (next I2C read performed by master will read this byte).
- 0x81-0xFF - Transmission completed and acknowledged; respond with i2c_write() to pre-load the transmit buffer for the next transaction (the next I2C read performed by master will read this byte).

**Function:** Returns the state of I2C communications in I2C slave mode after an SSP interrupt. The return value increments with each byte received or sent.

If 0x00 or 0x80 is returned, an i2c_read() needs to be performed to read the I2C address that was sent (it will match the address configured by #USE I2C so this value can be ignored)

**Availability:** Devices with i2c hardware

**Requires:** #USE I2C

**Examples:**

```c
#INT_SSP

void i2c_isr() {
    state = i2c_isr_state();
    if(state== 0 ) i2c_read();
    i@c_read();
    if(state == 0x80)
        i2c_read(2);
    if(state >= 0x80)
        i2c_write(send_buffer[state - 0x80]);
    else if(state > 0)
```
### i2c_poll()

**Syntax:**
```
i2c_poll()
i2c_poll(stream)
```

**Parameters:**
- `stream` (optional) - specify the stream defined in `#USE I2C`

**Returns:**
1 (TRUE) or 0 (FALSE)

**Function:**
The `I2C_POLL()` function should only be used when the built-in SSP is used. This function returns TRUE if the hardware has a received byte in the buffer. When a TRUE is returned, a call to `I2C_READ()` will immediately return the byte that was received.

**Availability:**
Devices with built-in I2C

**Requires:**
`#USE I2C`

**Examples:**
```
if(i2c_poll())
buffer [index]=i2c_read(); // read data
```

---

### i2c_read()

**Syntax:**
```
data = i2c_read();
data = i2c_read(ack);
data = i2c_read(stream, ack);
```
Parameters:  

- **ack**: Optional, defaults to 1.  
  0 indicates do not ack.  
  1 indicates to ack.  
  2 slave only, indicates to not release clock at end of read. Use when i2c_isr_state () returns 0x80.  

- **stream**: specify the stream defined in #USE I2C

Returns:  

- data - 8 bit int

Function:  

Reads a byte over the I2C interface. In master mode this function will generate the clock and in slave mode it will wait for the clock. There is no timeout for the slave, use i2c_poll() to prevent a lockup. Use restart_wdt() in the #USE I2C to strobe the watch-dog timer in the slave mode while waiting.

Availability:  

All devices.

Requires:  

#USE I2C

Examples:  

```c
i2c_start();
i2c_write(0x01);
data1 = i2c_read(TRUE);
data2 = i2c_read(FALSE);
i2c_stop();
```

Example Files:  

ex_extee.c with 2416.c

Also See:  

i2c_poll, i2c_speed, i2c_start, i2c_stop, i2c_slaveaddr, i2c_isr_state, i2c_write, #USE I2C, I2C Overview

---

### i2c_slaveaddr( )

Syntax:  

```c
I2C_SlaveAddr(addr);
I2C_SlaveAddr(stream, addr);
```

Parameters:  

- **addr**: 8 bit device address  
- **stream** (optional) - specifies the stream used in #USE I2C

Returns:  

Nothing

Function:  

This functions sets the address for the I2C interface in slave mode.
### i2c_speed( )

**Syntax:**
- `i2c_speed(baud)`
- `i2c_speed(stream, baud)`

**Parameters:**
- `baud` is the number of bits per second.
- `stream` - specify the stream defined in #USE I2C

**Returns:**
Nothing.

**Function:**
This function changes the I2c bit rate at run time. This only works if the hardware I2C module is being used.

**Availability:**
All devices.

**Requires:**
#USE I2C

**Examples:**
- `I2C_Speed(400000);`

**Example Files:**
- none

**Also See:**
- `i2c_poll`, `i2c_speed`, `i2c_start`, `i2c_stop`, `i2c_isr_state`, `i2c_write`, `i2c_read`, #USE I2C, I2C Overview
**i2c_start( )**

**Syntax:**

i2c_start()

i2c_start(stream)

i2c_start(stream, restart)

**Parameters:**

- **stream:** specify the stream defined in #USE I2C
- **restart:**
  - 2 – new restart is forced instead of start
  - 1 – normal start is performed
  - 0 (or not specified) – restart is done only if the compiler last encountered a I2C_START and no I2C_STOP

**Returns:** undefined

**Function:** Issues a start condition when in the I2C master mode. After the start condition the clock is held low until I2C_WRITE() is called. If another I2C_start is called in the same function before an i2c_stop is called, then a special restart condition is issued. Note that specific I2C protocol depends on the slave device. The I2C_START function will now accept an optional parameter. If 1 the compiler assumes the bus is in the stopped state. If 2 the compiler treats this I2C_START as a restart. If no parameter is passed a 2 is used only if the compiler compiled a I2C_START last with no I2C_STOP since.

**Availability:** All devices.

**Requires:** #USE I2C

**Examples:**

i2c_start();

i2c_write(0xa0); // Device address

i2c_write(address); // Data to device

i2c_start(); // Restart

i2c_write(0xa1); // to change data direction

data=i2c_read(0); // Now read from slave

i2c_stop();

**Example Files:** ex_extee.c with 2416.c

**Also See:** i2c_poll, i2c_speed, i2c_stop, i2c_slaveaddr, i2c_isr_state, i2c_write, i2c_read, #USE I2C, I2C Overview
# Built-in Functions

## i2c_stop( )

**Syntax:**

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2c_stop()</td>
</tr>
<tr>
<td>i2c_stop(stream)</td>
</tr>
</tbody>
</table>

**Parameters:**

| stream: (optional) specify stream defined in #USE I2C |

**Returns:**

undefined

**Function:**

Issues a stop condition when in the I2C master mode.

**Availability:**

All devices.

**Requires:**

#USE I2C

**Examples:**

```c
i2c_start(); // Start condition
i2c_write(0xa0); // Device address
i2c_write(5); // Device command
i2c_write(12); // Device data
i2c_stop(); // Stop condition
```

**Example Files:**

ex_extee.c with 2416.c

**Also See:**

i2c_poll, i2c_speed, i2c_start, i2c_slaveaddr, i2c_isr_state, i2c_write, i2c_read, #USE I2C, I2C Overview

## i2c_write( )

**Syntax:**

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2c_write (data)</td>
</tr>
<tr>
<td>i2c_write (stream, data)</td>
</tr>
</tbody>
</table>

**Parameters:**

<table>
<thead>
<tr>
<th>data is an 8 bit int</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream - specify the stream defined in #USE I2C</td>
</tr>
</tbody>
</table>

**Returns:**

This function returns the ACK Bit.
0 means ACK, 1 means NO ACK, 2 means there was a collision if in Multi_Master Mode.
This does not return an ACK if using i2c in slave mode.

**Function:**

Sends a single byte over the I2C interface. In master mode this function will generate a clock with the data and in slave mode it will wait for the clock.
from the master. No automatic timeout is provided in this function. This function returns the ACK bit. The LSB of the first write after a start determines the direction of data transfer (0 is master to slave). Note that specific I2C protocol depends on the slave device.

**Availability:** All devices.

**Requires:** #USE I2C

**Examples:**

```c
long cmd;
...
i2c_start(); // Start condition
i2c_write(0xa0);// Device address
i2c_write(cmd);// Low byte of command
i2c_write(cmd>>8);// High byte of command
i2c_stop(); // Stop condition
```

**Example Files:** ex_extee.c with 2416.c

**Also See:** i2c_poll, i2c_speed, i2c_start, i2c_stop, i2c_slaveaddr, i2c_isr_state, i2c_read, #USE I2C, I2C Overview

---

### input()

**Syntax:**

```c
value = input (pin)
```

**Parameters:**

`Pin` to read. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2 ) bit 3 would have a value of 0x2C2*8+3 or 5651 . This is defined as follows:

```c
#define PIN_A3 5651
```

The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. note that doing I/O with a variable instead of a constant will take much longer time.

**Returns:**

0 (or FALSE) if the pin is low,
1 (or TRUE) if the pin is high

**Function:**

This function returns the state of the indicated pin. The method of I/O is dependent on the last USE * _IO directive. By default with standard I/O before the input is done the data direction is set to input.

**Availability:** All devices.
Built-in Functions

**Requires:**
Pin constants are defined in the devices .h file

**Examples:**
```
while ( !input(PIN_B1) );
// waits for B1 to go high

if( input(PIN_A0) )
    printf("A0 is now high\r\n");

int16 i=PIN_B1;
while(!i);
// waits for B1 to go high
```

**Example Files:**
ex_pulse.c

**Also See:** input_x(), output_low(), output_high(), #USE FIXED_IO, #USE FAST_IO, #USE STANDARD_IO, General Purpose I/O

---

**input_change_x( )**

**Syntax:**
```
value = input_change_a();
value = input_change_b();
value = input_change_c();
value = input_change_d();
value = input_change_e();
value = input_change_f();
value = input_change_g();
value = input_change_h();
value = input_change_j();
value = input_change_k();
```

**Parameters:** None

**Returns:** An 8-bit or 16-bit int representing the changes on the port.

**Function:** This function reads the level of the pins on the port and compares them to the results the last time the input_change_x( ) function was called. A 1 is returned if the value has changed, 0 if the value is unchanged.

**Availability:** All devices.
**Requires:** None

**Examples:**

```
pin_check = input_change_b( );
```

**Example Files:** None

**Also See:** 

`input( ), input_x( ), output_x(), #USE FIXED_IO, #USE FAST_IO, #USE STANDARD_IO, General Purpose I/O`

---

**input_state( )**

**Syntax:**

```
value = input_state(pin)
```

**Parameters:** `pin` to read. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2 ) bit 3 would have a value of 0x2C2*8+3 or 5651 . This is defined as follows: #define PIN_A3 5651 .

**Returns:** Bit specifying whether pin is high or low. A 1 indicates the pin is high and a 0 indicates it is low.

**Function:** This function reads the level of a pin without changing the direction of the pin as INPUT() does.

**Availability:** All devices.

**Requires:** Nothing

**Examples:**

```
level = input_state(pin_A3);
printf("level: %d", level);
```

**Example Files:** None

**Also See:**

`input(), set_tris_x(), output_low(), output_high(), General Purpose I/O`
**input_x()**

**Syntax:**

```cpp
value = input_a()
value = input_b()
value = input_c()
value = input_d()
value = input_e()
value = input_f()
value = input_g()
value = input_h()
value = input_i()
value = input_k()
```

**Parameters:** None

**Returns:** An 16 bit int representing the port input data.

**Function:** Inputs an entire word from a port. The direction register is changed in accordance with the last specified #USE * IO directive. By default with standard I/O before the input is done the data direction is set to input.

**Availability:** All devices.

**Requires:** Nothing

**Examples:**

```cpp
data = input_b();
```

**Example Files:** `ex_psp.c`

**Also See:** `input()`, `output_x()`, `#USE FIXED_IO`, `#USE FAST_IO`, `#USE STANDARD_IO`

---

**interrupt_active()**

**Syntax:**

```cpp
interrupt_active (interrupt)
```

**Parameters:** Interrupt – constant specifying the interrupt

**Returns:** Boolean value
**Function:** The function checks the interrupt flag of the specified interrupt and returns true in case the flag is set.

**Availability:** Device with interrupts

**Requires:** Should have a #INTxxxx, Constants are defined in the devices .h file.

**Examples:**
- `interrupt_active(INT_TIMER0);`
- `interrupt_active(INT_TIMER1);`

**Example Files:** None

**Also See:** `disable_interrupts()`, #INT, Interrupts Overview, clear_interrupt, enable_interrupts()

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum(char)</td>
<td>checks if the character is alphanumeric</td>
</tr>
<tr>
<td>isalpha(char)</td>
<td>checks if the character is a letter</td>
</tr>
<tr>
<td>isdigit(char)</td>
<td>checks if the character is a digit</td>
</tr>
<tr>
<td>islower(char)</td>
<td>checks if the character is lowercase</td>
</tr>
<tr>
<td>isspace(char)</td>
<td>checks if the character is a space</td>
</tr>
<tr>
<td>isupper(char)</td>
<td>checks if the character is uppercase</td>
</tr>
<tr>
<td>isxdigit(char)</td>
<td>checks if the character is a digit</td>
</tr>
<tr>
<td>isprint(char)</td>
<td>checks if the character is printable</td>
</tr>
<tr>
<td>ispunct(char)</td>
<td>checks if the character is punctuation</td>
</tr>
</tbody>
</table>

**Syntax:**

```c
value = isalnum(datac)
value = isalpha(datac)
value = isdigit(datac)
value = islower(datac)
value = isspace(datac)
value = isupper(datac)
value = isxdigit(datac)
value = isprint(datac)
value = punct(datac)
```

**Parameters:**

- `datac` is a 8 bit character

**Returns:**

- 0 (or FALSE) if `datac` does not match the criteria
- 1 (or TRUE) if `datac` does match the criteria.
### Built-in Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Tests a character to see if it meets specific criteria as follows:</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum(x)</td>
<td>$X$ is 0..9, 'A'..'Z', or 'a'..'z'</td>
</tr>
<tr>
<td>isalpha(x)</td>
<td>$X$ is 'A'..'Z' or 'a'..'z'</td>
</tr>
<tr>
<td>isdigit(x)</td>
<td>$X$ is '0'..'9'</td>
</tr>
<tr>
<td>islower(x)</td>
<td>$X$ is 'a'..'z'</td>
</tr>
<tr>
<td>isupper(x)</td>
<td>$X$ is 'A'..'Z'</td>
</tr>
<tr>
<td>isspace(x)</td>
<td>$X$ is a space</td>
</tr>
<tr>
<td>isxdigit(x)</td>
<td>$X$ is '0'..'9', 'A'..'F', or 'a'..'f'</td>
</tr>
<tr>
<td>iscntrl(x)</td>
<td>$X$ is less than a space</td>
</tr>
<tr>
<td>isgraph(x)</td>
<td>$X$ is greater than a space</td>
</tr>
<tr>
<td>isprint(x)</td>
<td>$X$ is greater than or equal to a space</td>
</tr>
<tr>
<td>ispunct(x)</td>
<td>$X$ is greater than a space and not a letter or number</td>
</tr>
</tbody>
</table>

**Availability:** All devices.

**Requires:** #INCLUDE <ctype.h>

**Examples:**
```c
char id[20];
...
if(isalpha(id[0])) {
    valid_id=TRUE;
    for(i=1;i<strlen(id);i++)
        valid_id=valid_id && isalnum(id[i]);
} else
    valid_id=FALSE;
```

**Example Files:** ex_str.c

**Also See:** isamong()

---

### isamong( )

**Syntax:**

```
result = isamong (value, cstring)
```

**Parameters:**

- **value** is a character
- **cstring** is a constant sting

**Returns:**

- 0 (or FALSE) if value is not in cstring
- 1 (or TRUE) if value is in cstring

**Function:** Returns TRUE if a character is one of the characters in a constant string.
Availability: All devices

Requires: Nothing

Examples: char x = 'x';
...
if ( isamong ( x, "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ") )
    printf ("The character is valid");

Example Files:

#include <ctype.h>

Also See: isalnum(), isalpha(), isdigit(), isspace(), islower(), isupper(), isxdigit()

### itoa()

Syntax: string = itoa(i32value, i8base, string)
string = itoa(i48value, i8base, string)
string = itoa(i64value, i8base, string)

Parameters: i32value is a 32 bit int
i48value is a 48 bit int
i64value is a 64 bit int
i8base is a 8 bit int
string is a pointer to a null terminated string of characters

Returns: string is a pointer to a null terminated string of characters

Function: Converts the signed int32, int48, or a int64 to a string according to the provided base and returns the converted value if any. If the result cannot be represented, the function will return 0.

Availability: All devices

Requires: #include <stdlib.h>

Examples: int32 x=1234;
char string[5];

itoa(x,10, string);
// string is now “1234”
**Example Files:**
None

**Also See:**
None

---

### kbhit()

**Syntax:**

```plaintext
value = kbhit()
value = kbhit(stream)
```

**Parameters:**

*stream* is the stream id assigned to an available RS232 port. If the stream parameter is not included, the function uses the primary stream used by `getc()`.

**Returns:**

0 (or FALSE) if `getc()` will need to wait for a character to come in, 1 (or TRUE) if a character is ready for `getc()`

**Function:**

If the RS232 is under software control this function returns TRUE if the start bit of a character is being sent on the RS232 RCV pin. If the RS232 is hardware this function returns TRUE if a character has been received and is waiting in the hardware buffer for `getc()` to read. This function may be used to poll for data without stopping and waiting for the data to appear. Note that in the case of software RS232 this function should be called at least 10 times the bit rate to ensure incoming data is not lost.

**Availability:**

All devices.

**Requires:**

`#USE RS232`

**Examples:**

```plaintext
char timed_getc() {
    long timeout;
    timeout_error=FALSE;
    timeout=0;
    while(!kbhit()&&(++timeout<50000)) // 1/2 second
        delay_us(10);
    if(kbhit())
        return(getc());
    else {
        timeout_error=TRUE;
    }
}
```
Example Files: ex_tgetc.c
Also See: getc(), #USE RS232, RS232 I/O Overview

---

### label_address( )

**Syntax:**

```c
value = label_address(label);
```

**Parameters:**

- `label` is a C label anywhere in the function

**Returns:**

A 16 bit int in PCB,PCM and a 32 bit int for PCH, PCD

**Function:**

This function obtains the address in ROM of the next instruction after the label. This is not a normally used function except in very special situations.

**Availability:**

All devices.

**Requires:**

Nothing

**Examples:**

```c
start:
    a = (b+c)<<2;
end:
    printf("It takes %lu ROM locations.\r\n", label_address(end) - label_address(start));
```

Example Files: setjmp.h
Also See: goto_address()

---

### labs( )

**Syntax:**

```c
result = labs(value)
```

**Parameters:**

- `value` is a 16, 32, 48 or 64 bit signed long int

**Returns:**

A signed long int of type `value`
**Function:** Computes the absolute value of a long integer.

**Availability:** All devices.

**Requires:** `#INCLUDE <stdlib.h>`

**Examples:**
```c
if(labs( target_value - actual_value ) > 500)
  printf("Error is over 500 points\n");
```

**Example Files:** None

**Also See:** `abs()`

---

**ldexp( )**

**Syntax:**
```c
result= ldexp (value, exp);
```

**Parameters:**
- `value` is float any float type
- `exp` is a signed int.

**Returns:**
result is a float with value `result` times 2 raised to power `exp`. `result` will have a precision equal to `value`

**Function:**
The ldexp function multiplies a floating-point number by an integral power of 2.

**Availability:** All devices.

**Requires:** `#INCLUDE <math.h>`

**Examples:**
```c
float result;
result=ldexp(.5,0);
// result is .5
```

**Example Files:** None

**Also See:** `frexp()`, `exp()`, `log()`, `log10()`, `modf()`
### log( )

**Syntax:**  
\[ \text{result} = \log (\text{value}) \]

**Parameters:**  
`value` is any float type

**Returns:**  
A float with precision equal to `value`

**Function:**  
Computes the natural logarithm of the float x. If the argument is less than or equal to zero or too large, the behavior is undefined.

Note on error handling:  
"errno.h" is included then the domain and range errors are stored in the `errno` variable. The user can check the `errno` to see if an error has occurred and print the error using the `perror` function.

Domain error occurs in the following cases:  
• log: when the argument is negative

**Availability:**  
All devices

**Requires:**  
`#INCLUDE <math.h>`

**Examples:**  
\[ \text{ln}x = \log(x); \]

**Example Files:**  
None

**Also See:**  
`log10()`, `exp()`, `pow()`

---

### log10( )

**Syntax:**  
\[ \text{result} = \log10 (\text{value}) \]

**Parameters:**  
`value` is any float type

**Returns:**  
A float with precision equal to `value`

**Function:**  
Computes the base-ten logarithm of the float x. If the argument is less than or equal to zero or too large, the behavior is undefined.
Built-in Functions

Note on error handling:
If "errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Domain error occurs in the following cases:
• log10: when the argument is negative

Availability: All devices

Requires: #INCLUDE <math.h>

Examples:
```
db = log10( read_adc() *(5.0/255) ) *10;
```

Example Files: None

Also See: log(), exp(), pow()

---

**longjmp( )**

**Syntax:**
```
longjmp (env, val)
```

**Parameters:**
- **env:** The data object that will be restored by this function
- **val:** The value that the function setjmp will return. If val is 0 then the function setjmp will return 1 instead.

**Returns:**
After longjmp is completed, program execution continues as if the corresponding invocation of the setjmp function had just returned the value specified by val.

**Function:**
Performs the non-local transfer of control.

Availability: All devices

Requires: #INCLUDE <setjmp.h>

Examples:
```
longjmp(jmpbuf, 1);
```

Example Files: None

Also See: setjmp()
make8( )

Syntax: \[ i8 = \text{MAKE8}(\text{var}, \text{offset}) \]

Parameters: \text{var} is a 16 or 32 bit integer. 
\text{offset} is a byte offset of 0,1,2 or 3.

Returns: An 8 bit integer

Function: Extracts the byte at offset from var. Same as: \[ i8 = (((\text{var} >> (\text{offset} \times 8)) \& 0xff) \]
except it is done with a single byte move.

Availability: All devices

Requires: Nothing

Examples:
\begin{verbatim}
int32 x;
int y;

y = make8(x,3); // Gets MSB of x
\end{verbatim}

Example Files: None

Also See: make16(), make32()

make16( )

Syntax: \[ i16 = \text{MAKE16}(\text{varhigh}, \text{varlow}) \]

Parameters: \text{varhigh} and \text{varlow} are 8 bit integers.

Returns: A 16 bit integer

Function: Makes a 16 bit number out of two 8 bit numbers. If either parameter is 16 or 32 bits only the lsb is used. Same as: \[ i16 = (\text{int16})(\text{varhigh} \& 0xff) \times 100 + (\text{varlow} \& 0xff) \]
except it is done with two byte moves.

Availability: All devices

Requires: Nothing
**Built-in Functions**

**Examples:**
```c
long x;
int hi,lo;

x = make16(hi,lo);
```

**Example Files:** ltc1298.c

**Also See:** make8(), make32()

---

### make32()

**Syntax:**
```c
i32 = MAKE32(var1, var2, var3, var4)
```

**Parameters:**
- `var1-4` are a 8 or 16 bit integers. `var2-4` are optional.

**Returns:**
A 32 bit integer

**Function:**
Makes a 32 bit number out of any combination of 8 and 16 bit numbers. Note that the number of parameters may be 1 to 4. The msb is first. If the total bits provided is less than 32 then zeros are added at the msb.

**Availability:**
All devices

**Requires:**
Nothing

**Examples:**
```c
int32 x;
int y;
long z;

x = make32(1,2,3,4); // x is 0x01020304

y=0x12;
z=0x4321;

x = make32(y,z); // x is 0x00124321
x = make32(y,y,z); // x is 0x12124321
```

**Example Files:** ex_freqc.c

**Also See:** make8(), make16()
### malloc()

**Syntax:**

```c
ptr=malloc(size)
```

**Parameters:**

- `size` is an integer representing the number of bytes to be allocated.

**Returns:**

- A pointer to the allocated memory, if any. Returns null otherwise.

**Function:**

- The `malloc` function allocates space for an object whose size is specified by `size` and whose value is indeterminate.

**Availability:**

- All devices

**Requires:**

- `#INCLUDE <stdlib.h>`

**Examples:**

```c
int * iptr;
iptr=malloc(10);
// iptr will point to a block of memory of 10 bytes.
```

**Example Files:**

- None

**Also See:**

- `realloc()`, `free()`, `calloc()`

### memcpy()  memmove()

**Syntax:**

- `memcpy (destination, source, n)`
- `memmove (destination, source, n)`

**Parameters:**

- `destination` is a pointer to the destination memory.
- `source` is a pointer to the source memory.
- `n` is the number of bytes to transfer

**Returns:**

- undefined

**Function:**

- Copies `n` bytes from `source` to `destination` in RAM. Be aware that array names are pointers where other variable names and structure names are not (and therefore need a & before them).

- Memmove performs a safe copy (overlapping objects doesn't cause a problem). Copying takes place as if the `n` characters from the source are
first copied into a temporary array of n characters that doesn't overlap the destination and source objects. Then the n characters from the temporary array are copied to destination.

**Availability:** All devices

**Requires:** Nothing

**Examples:**
- `memcpy(&structA, &structB, sizeof (structA));`
- `memcpy(arrayA, arrayB, sizeof (arrayA));`
- `memcpy(&structA, &databyte, 1);`

```c
char a[20]="hello";
memmove(a,a+2,5);
// a is now "llo"
```

**Example Files:** None

**Also See:** `strcpy()`, `memset()`

---

### `memset()`

**Syntax:**

```c
memset (destination, value, n)
```

**Parameters:**

- `destination` is a pointer to memory.
- `value` is a 8 bit int
- `n` is a 16 bit int.

**Returns:** undefined

**Function:** Sets n number of bytes, starting at destination, to value. Be aware that array names are pointers where other variable names and structure names are not (and therefore need a & before them).

**Availability:** All devices

**Requires:** Nothing

**Examples:**
- `memset(arrayA, 0, sizeof(arrayA));`
- `memset(arrayB, '?', sizeof(arrayB));`
- `memset(&structA, 0xFF, sizeof(structA));`
**modf()**

**Syntax:**
\[
\text{result} = \text{modf}(\text{value}, \& \text{integral})
\]

**Parameters:**
- **value** is any float type
- **integral** is any float type

**Returns:**
result is a float with precision equal to value

**Function:**
The modf function breaks the argument value into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a float in the object integral.

**Availability:**
All devices

**Requires:**
#include <math.h>

**Examples:**
```c
float result, integral;
result = modf(123.987, &integral);
// result is .987 and integral is 123.0000
```

**_mul()**

**Syntax:**
```c
prod = _mul(val1, val2);
```

**Parameters:**
- **val1** and **val2** are both 8-bit, 16-bit, or 48-bit integers

**Returns:**
<table>
<thead>
<tr>
<th>val1</th>
<th>val2</th>
<th>prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>16*</td>
<td>16</td>
<td>32</td>
</tr>
</tbody>
</table>
Built-in Functions

<table>
<thead>
<tr>
<th>32*</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>48*</td>
<td>48</td>
<td>64**</td>
</tr>
</tbody>
</table>

* or less
** large numbers will overflow with wrong results

Function: Performs an optimized multiplication. By accepting a different type than it returns, this function avoids the overhead of converting the parameters to a larger type.

Availability: All devices

Requires: Nothing

Examples:

```c
int a=50, b=100;
long int c;
c = _mul(a, b);     //c holds 5000
```

Example Files: None

Also See: None

nargs( )

Syntax: `void foo(char * str, int count, ...)`

Parameters: The function can take variable parameters. The user can use stdarg library to create functions that take variable parameters.

Returns: Function dependent.

Function: The stdarg library allows the user to create functions that supports variable arguments. The function that will accept a variable number of arguments must have at least one actual, known parameters, and it may have more. The number of arguments is often passed to the function in one of its actual parameters. If the variable-length argument list can involve more that one type, the type information is generally passed as well. Before processing can begin, the function creates a special argument pointer of type va_list.

Availability: All devices
Requires:  

#INCLUDE <stdarg.h>

Examples:  

```c
int foo(int num, ...)
{
    int sum = 0;
    int i;
    va_list argptr; // create special argument pointer
    va_start(argptr,num); // initialize argptr
    for(i=0; i<num; i++)
        sum = sum + va_arg(argptr, int);
    va_end(argptr);   // end variable processing
    return sum;
}

void main()
{
    int total;
    total = foo(2,4,6,9,10,2);
}
```

Example Files:  

None

Also See:  

va_start(), va_end(), va_arg()

---

**offsetof( ) offsetofbit( )**

Syntax:  

```c
value = offsetof(stype, field);
value = offsetofbit(stype, field);
```

Parameters:  

*stype* is a structure type name.  
*Field* is a field from the above structure

Returns:  

An 8 bit byte

Function:  

These functions return an offset into a structure for the indicated field.  
offsetof returns the offset in bytes and offsetofbit returns the offset in bits.

Availability:  

All devices

Requires:  

#INCLUDE <stddef.h>

Examples:  

```c
struct time_structure {
```
```c
int hour, min, sec;
int zone : 4;
int1 daylight_savings;

x = offsetof(time_structure, sec);
// x will be 2
x = offsetofbit(time_structure, sec);
// x will be 16
x = offsetof(time_structure,
    daylight_savings);
// x will be 3
x = offsetofbit(time_structure,
    daylight_savings);
// x will be 28
```

Example Files:
None

Also See:
None

### output_x( )

**Syntax:**
- `output_a (value)`
- `output_b (value)`
- `output_c (value)`
- `output_d (value)`
- `output_e (value)`
- `output_f (value)`
- `output_g (value)`
- `output_h (value)`
- `output_j (value)`
- `output_k (value)`

**Parameters:**
`value` is a 16 bit int

**Returns:**
undefined

**Function:**
Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

**Availability:**
All devices, however not all devices have all ports (A-E)

**Requires:**
Nothing
Examples: OUTPUT_B(0xf0);

Example Files: ex_patg.c

Also See: input(), output_low(), output_high(), output_float(), output_bit(), #USE FIXED_IO, #USE FAST_IO, #USE STANDARD_IO, General Purpose I/O

output_bit( )

Syntax: output_bit (pin, value)

Parameters: Pins are defined in the devices .h file. The actual number is a bit address. For example, port a (byte 0x2C2 ) bit 3 would have a value of 0x2C2*8+3 or 5651 . This is defined as follows: #define PIN_A3 5651 . The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time. Value is a 1 or a 0.

Returns: undefined

Function: Outputs the specified value (0 or 1) to the specified I/O pin. The method of setting the direction register is determined by the last #USE *_IO directive.

Availability: All devices.

Requires: Pin constants are defined in the devices .h file

Examples: output_bit( PIN_B0, 0);
    // Same as output_low(pin_B0);

    output_bit( PIN_B0,input( PIN_B1 ) );
    // Make pin B0 the same as B1

    output_bit( PIN_B0,shift_left(&data,1,input(PIN_B1))); // Output the MSB of data to
    // B0 and at the same time
    // shift B1 into the LSB of data
```c
int16 i=PIN_B0;
output_bit(i,shift_left(&data,1,input(PIN_B1)));
//same as above example, but
//uses a variable instead of a constant
```

### Example Files:
- `ex_extee.c` with `9356.c`

### Also See:
- `input()`, `output_low()`, `output_high()`, `output_float()`, `output_x()`.
- `#USE_FIXED_IO`, `#USE_FAST_IO`, `#USE_STANDARD_IO`, `General Purpose I/O`

---

## `output_drive()`

### Syntax:

```
output_drive(pin)
```

### Parameters:
- **Pins** are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651. This is defined as follows: `#DEFINE PIN_A3 5651`.

### Returns:
- undefined

### Function:
- Sets the specified pin to the output mode.

### Availability:
- All devices.

### Requires:
- Pin constants are defined in the devices.h file.

### Examples:
- `output_drive(pin_A0);` // sets pin_A0 to output its value
- `output_bit(pin_B0, input(pin_A0))` // makes B0 the same as A0

### Example Files:
- None

### Also See:
- `input()`, `output_low()`, `output_high()`, `output_bit()`, `output_x()`, `output_float()`
output_float( )

Syntax: output_float (pin)

Parameters: Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2 ) bit 3 would have a value of 0x2C2*8+3 or 5651 . This is defined as follows: #DEFINE PIN_A3 5651 . The PIN could also be a variable to identify the pin. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. Note that doing I/O with a variable instead of a constant will take much longer time.

Returns: undefined

Function: Sets the specified pin to the input mode. This will allow the pin to float high to represent a high on an open collector type of connection.

Availability: All devices.

Requires: Pin constants are defined in the devices .h file

Examples: if( (data & 0x80)==0 )
            output_low(pin_A0);
        else
            output_float(pin_A0);

Example Files: None

Also See: input(), output_low(), output_high(), output_bit(), output_xf(), output_drive(), #USE FIXED_IO, #USE FAST_IO, #USE STANDARD_IO, General Purpose I/O

output_high( )

Syntax: output_high (pin)

Parameters: Pin to write to. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2 ) bit 3 would have a value of 0x2C2*8+3 or 5651 . This is defined as follows: #DEFINE PIN_A3 5651 . The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate...
register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time.

Returns: undefined

Function: Sets a given pin to the high state. The method of I/O used is dependent on the last USE *_IO directive.

Availability: All devices.

Requires: Pin constants are defined in the devices .h file

Examples: output_high(PIN_A0);
output_low(PIN_A1);

Example Files: ex_sqw.c

Also See: input(), output_low(), output_float(), output_bit(), output_x(), #USE FIXED_IO, #USE FAST_IO, #USE STANDARD_IO, General Purpose I/O

output_low( )

Syntax: output_low (pin)

Parameters: Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651. This is defined as follows: #DEFINE PIN_A3 5651. The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time.

Returns: undefined

Function: Sets a given pin to the ground state. The method of I/O used is dependent on the last USE *_IO directive.

Availability: All devices.

Requires: Pin constants are defined in the devices .h file
<table>
<thead>
<tr>
<th>Examples:</th>
<th>output_low(PIN_A0);</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Int16i=PIN_A1;</td>
</tr>
<tr>
<td></td>
<td>output_low(PIN_A1);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example Files:</th>
<th>ex_sqw.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Also See:</td>
<td>input(), output_high(), output_float(), output_bit(), output_x(), #USE FIXED_IO, #USE FAST_IO, #USE STANDARD_IO, General Purpose I/O</td>
</tr>
</tbody>
</table>

---

### output_toggle()

**Syntax:**

```
output_toggle(pin)
```

**Parameters:**

Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651. This is defined as follows: #DEFINE PIN_A3 5651.

**Returns:**

Undefined

**Function:**

Toggles the high/low state of the specified pin.

**Availability:**

All devices.

**Requires:**

Pin constants are defined in the devices .h file

**Examples:**

```
output_toggle(PIN_B4);
```

<table>
<thead>
<tr>
<th>Example Files:</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Also See:</td>
<td>input(), output_high(), output_low(), output_bit(), output_x()</td>
</tr>
</tbody>
</table>

---

### perror()

**Syntax:**

```
perror(string);
```

**Parameters:**

`string` is a constant string or array of characters (null terminated).

**Returns:**

Nothing
<table>
<thead>
<tr>
<th>Function:</th>
<th>This function prints out to STDERR the supplied string and a description of the last system error (usually a math error).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability:</td>
<td>All devices.</td>
</tr>
<tr>
<td>Requires:</td>
<td>#USE RS232, #INCLUDE &lt;errno.h&gt;</td>
</tr>
</tbody>
</table>
| Examples: | x = sin(y);  
if(errno!=0)  
perror("Problem in find_area"); |

**Example Files:** None  
**Also See:** RS232 I/O Overview

### pid_busy()

**Syntax:**

result = pid_busy();

**Parameters:** None

**Returns:** TRUE if PID module is busy or FALSE is PID module is not busy.

**Function:** To check if the PID module is busy with a calculation.

**Availability:** All devices with a PID module.

**Requires:** Nothing

**Examples:**

```c
pid_get_result(PID_START_ONLY, ADCResult);
while(pid_busy());
pid_get_result(PID_READ_ONLY, &PIDResult);
```

**Example Files:** None  
**Also See:** setup_pid(), pid_write(), pid_get_result(), pid_read()
**pid_get_result( )**

**Syntax:**

```c
pid_get_result(set_point, input, &output);  //Start and Read  
pid_get_result(mode, set_point, input);    //Start Only  
pid_get_result(mode, &output)              //Read Only  
pid_get_result(mode, set_point, input, &output);
```

**Parameters:**

- **mode** - constant parameter specifying whether to only start the calculation, only read the result, or start the calculation and read the result. The options are defined in the device's header file as:
  - PID_START_READ
  - PID_READ_ONLY
  - PID_START_ONLY

- **set_point** - a 16-bit variable or constant representing the set point of the control system, the value the input from the control system is compared against to determine the error in the system.

- **input** - a 16-bit variable or constant representing the input from the control system.

- **output** - a structure that the output of the PID module will be saved to. Either pass the address of the structure as the parameter, or a pointer to the structure as the parameter.

**Returns:**

Nothing

**Function:**

To pass the set point and input from the control system to the PID module, start the PID calculation and get the result of the PID calculation. The PID calculation starts, automatically when the input is written to the PID module's input registers.

**Availability:**

All devices with a PID module.

**Requires:**

Constants are defined in the device's .h file.

**Examples:**

```c
pid_get_result(SetPoint, ADCResult, &PIDOutput);  //Start and Read  
pid_get_result(PID_START_ONLY, SetPoint, ADCResult);  //Start Only  
pid_get_result(PID_READ_ONLY, &PIDResult);  //Read Only
```

**Example**

None
Files: setup_pid(), pid_read(), pid_write(), pid_busy()

pid_read()

Syntax:  pid_read(register, &output);

Parameters:  register- constant specifying which PID registers to read. The registers that can be written are defined in the device's header file as:
  - PID_ADDR_ACCUMULATOR
  - PID_ADDR_OUTPUT
  - PID_ADDR_Z1
  - PID_ADDR_Z2
  - PID_ADDR_K1
  - PID_ADDR_K2
  - PID_ADDR_K3

output - a 16-bit variable, 32-bit variable or structure that specified PID registers value will be saved to. The size depends on the registers that are being read. Either pass the address of the variable or structure as the parameter, or a pointer to the variable or structure as the parameter.

Returns:  Nothing

Function:  To read the current value of the Accumulator, Output, Z1, Z2, Set Point, K1, K2 or K3 PID registers. If the PID is busy with a calculation the function will wait for module to finish calculation before reading the specified register.

Availability:  All devices with a PID module.

Requires:  Constants are defined in the device's .h file.

Examples:  pid_read(PID_ADDR_Z1, &value_z1);

Example Files:  None

Also See: setup_pid(), pid_write(), pid_get_result(), pid_busy()
## pid_write()

**Syntax:**

```c
pid_write(register, &input);
```

<table>
<thead>
<tr>
<th>Parameters:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>register</code></td>
<td>constant specifying which PID registers to write. The registers that can be written are defined in the device's header file as:</td>
</tr>
<tr>
<td></td>
<td>- PID_ADDR_ACCUMULATOR</td>
</tr>
<tr>
<td></td>
<td>- PID_ADDR_OUTPUT</td>
</tr>
<tr>
<td></td>
<td>- PID_ADDR_Z1</td>
</tr>
<tr>
<td></td>
<td>- PID_ADDR_Z2</td>
</tr>
<tr>
<td></td>
<td>- PID_ADDR_Z3</td>
</tr>
<tr>
<td></td>
<td>- PID_ADDR_K1</td>
</tr>
<tr>
<td></td>
<td>- PID_ADDR_K2</td>
</tr>
<tr>
<td></td>
<td>- PID_ADDR_K3</td>
</tr>
<tr>
<td><code>input</code></td>
<td>a 16-bit variable, 32-bit variable or structure that contains the data to be written. The size depends on the registers that are being written. Either pass the address of the variable or structure as the parameter, or a pointer to the variable or structure as the parameter.</td>
</tr>
</tbody>
</table>

| Returns:             | Nothing                                                                  |

| Function:            | To write a new value for the Accumulator, Output, Z1, Z2, Set Point, K1, K2 or K3 PID registers. If the PID is busy with a calculation the function will wait for module to finish the calculation before writing the specified register. |

| Availability:        | All devices with a PID module.                                           |

| Requires:            | Constants are defined in the device's .h file.                          |

| Examples:            | `pid_write(PID_ADDR_Z1, &value_z1);`                                     |

| Example Files:       | None                                                                     |

| Also See:            | `setup_pid()`, `pid_read()`, `pid_get_result()`, `pid_busy()`             |
### pll_locked()

<table>
<thead>
<tr>
<th>Syntax:</th>
<th>result=pll_locked();</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters:</td>
<td>None</td>
</tr>
<tr>
<td>Returns:</td>
<td>A short int. TRUE if the PLL is locked/ready, FALSE if PLL is not locked/ready</td>
</tr>
<tr>
<td>Function:</td>
<td>This function allows testing the PLL Ready Flag bit to determined if the PLL is stable and running.</td>
</tr>
<tr>
<td>Availability:</td>
<td>Devices with a Phase Locked Loop (PLL). Not all devices have a PLL Ready Flag, for those devices the pll_locked() function will always return TRUE.</td>
</tr>
<tr>
<td>Requires:</td>
<td>Nothing.</td>
</tr>
<tr>
<td>Examples:</td>
<td>while(!pll_locked());</td>
</tr>
<tr>
<td>Example Files:</td>
<td>None</td>
</tr>
<tr>
<td>Also See:</td>
<td>#use delay</td>
</tr>
</tbody>
</table>

### pmp_address(address)

<table>
<thead>
<tr>
<th>Syntax:</th>
<th>pmp_address(address);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters:</td>
<td>address: The address which is a 16 bit destination address value. This will setup the address register on the PMP module and is only used in Master mode.</td>
</tr>
<tr>
<td>Returns:</td>
<td>undefined</td>
</tr>
<tr>
<td>Function:</td>
<td>Configures the address register of the PMP module with the destination address during Master mode operation. The address can be either 14, 15 or 16 bits based on the multiplexing used for the Chip Select Lines 1 and 2.</td>
</tr>
<tr>
<td>Availability:</td>
<td>Only the devices with a built in Parallel Port module.</td>
</tr>
<tr>
<td>Requires:</td>
<td>Nothing.</td>
</tr>
</tbody>
</table>
Examples:

pmp_address( 0x2100); // Sets up Address register to 0x2100

Example Files:
None

Also See:
setup_pmp(), pmp_address(), pmp_read(), psp_read(), psp_write(),
pmp_write(), psp_output_full(), psp_input_full(), psp_overflow(),
pmp_output_full(), pmp_input_full(), pmp_overflow().
See header file for device selected.

pmp_output_full()  pmp_input_full()  pmp_overflow()  pmp_error()  pmp_timeout()

Syntax:
result = pmp_output_full()  //PMP only
result = pmp_input_full()    //PMP only
result = pmp_overflow()      //PMP only
result = pmp_error()         //EPMP only
result = pmp_timeout()       //EPMP only

Parameters:  None

Returns:  A 0 (FALSE) or 1 (TRUE)

Function:  These functions check the Parallel Port for the indicated conditions and return TRUE or FALSE.

Availability:  This function is only available on devices with Parallel Port hardware on chips.

Requires:  Nothing.

Examples:
while (pmp_output_full()) ;
pmp_data = command;
while(!pmp_input_full()) ;
if ( pmp_overflow() )
   error = TRUE;
else
   data = pmp_data;

Example  None
### pmp_read()

**Syntax:**

- `result = pmp_read();` // Parallel Master Port
- `result = pmp_read8(address);` // Enhanced Parallel Master Port
- `result = pmp_read16(address);` // Enhanced Parallel Master Port
- `pmp_read8(address,pointer,count);` // Enhanced Parallel Master Port
- `pmp_read16(address,pointer,count);` // Enhanced Parallel Master Port

**Parameters:**
- `address`: EPMP only, address in EDS memory that is mapped to address from parallel port device to read data from or start reading data from. (All address in EDS memory are word aligned)
- `pointer`: EPMP only, pointer to array to read data to.
- `count`: EPMP only, number of bytes to read. For `pmp_read16()` number of bytes must be even.

**Returns:**
For `pmp_read()`, `pmp_read8(address)` or `pmp_read16()` an 8 or 16 bit value. For `pmp_read8(address,pointer,count)` and `pmp_read16(address,pointer,count)` undefined.

**Function:**
For PMP module, this will read a byte from the next buffer location. For EPMP module, reads one byte/word or count bytes of data from the address mapped to the EDS memory location. The address is used in conjunction with the offset address set with the `setup_pmp_cs1()` and `setup_pmp_cs2()` functions to determine which address lines are high or low during the read.

**Availability:**
Only the devices with a built in Parallel Master Port module or an Enhanced Parallel Master Port module.

**Requires:**
Nothing.

**Examples:**

```c
result = pmp_read();  // PMP reads next byte of // data
```
result = pmp_read8(0x8000);  //EPMP reads byte of data
from the address mapped //to first address in
//EDS memory.
pmp_read16(0x8002,ptr,16);  //EPMP reads 16 bytes of
//data and returns to array
//pointed to by ptr
//starting at address mapped
//to address 0x8002 in
//EDS memory.

Example
Files:
None
Also See: setup_pmp(), setup_pmp_csx(), pmp_address(), pmp_read(), psp_read(), psp_write(), pmp_write(), psp_output_full(), psp_input_full(), psp_overflow(), pmp_output_full(), pmp_input_full(), pmp_overflow()
pmp_error(), pmp_timeout(), psp_error(), psp_timeout()

pmp_write( )

Syntax:

\[
\begin{align*}
\text{pmp\_write}(\text{data}); & \quad \text{//Parallel Master Port} \\
\text{pmp\_write8}(\text{address, data}); & \quad \text{//Enhanced Parallel Master Port} \\
\text{pmp\_write8}(\text{address, pointer, data}); & \quad \text{//Enhanced Parallel Master Port} \\
\text{pmp\_write16}(\text{address, data}); & \quad \text{//Enhanced Parallel Master Port} \\
\text{pmp\_write16}(\text{address, pointer, data}); & \quad \text{//Enhanced Parallel Master Port}
\end{align*}
\]

Parameters:

\[
\begin{align*}
\text{data} & \quad \text{- The byte of data to be written.} \\
\text{address} & \quad \text{- EPMP only, address in EDS memory that is mapped to} \\
& \text{address from parallel port device to write data to or start writing data to.} \\
& \text{(All addresses in EDS memory are word aligned)} \\
\text{pointer} & \quad \text{- EPMP only, pointer to data to be written} \\
\text{count} & \quad \text{- EPMP only, number of bytes to write. For pmp\_write16( ) number of} \\
& \text{bytes must be even.}
\end{align*}
\]

Returns:

Undefined.

Function:

For PMP modules, this will write a byte of data to the next buffer location.
For EPMP modules writes one byte/word or count bytes of data from the address mapped to the EDS memory location. The address is used in conjunction with the offset address set with the setup_pmp_cs1( ) and setup_pmp_cs2( ) functions to determine which address lines are high or low during write.

**Availability:** Only the devices with a built in Parallel Master Port module or Enhanced Parallel Master Port modules.

**Requires:** Nothing.

**Examples:**

```c
pmp_write( data );  //Write the data byte to
                    //the next buffer location.
pmp_write8(0x8000,data);  //EPMP writes the data byte to
                          //the address mapped to
                          //the first location in
                          //EDS memory.
pmp_write16(0x8002,ptr,16);  //EPMP writes 16 bytes of
                             //data pointed to by ptr
                             //starting at address mapped
                             //to address 0x8002 in
                             //EDS Memory
```

**Example Files:** None

**Also See:**

- setup_pmp()
- setup_pmp_csx()
- pmp_address()
- pmp_read()
- psp_read()
- psp_write()
- pmp_write()
- psp_output_full()
- psp_input_full()
- psp_overflow()
- pmp_output_full()
- pmp_input_full()
- pmp_overflow()
- pmp_error()
- pmp_timeout()
- psp_error()
- psp_timeout()

---

**port_x_pullups( )**

**Syntax:**

```c
port_a_pullups ( value )
port_b_pullups ( value )
port_d_pullups ( value )
port_e_pullups ( value )
port_j_pullups ( value )
port_x_pullups ( upmask )
port_x_pullups ( upmask, downmask )
```

**Parameters:**

- `value` is TRUE or FALSE on most parts, some parts that allow pullups to be specified on individual pins permit an 8 bit int here, one bit for each port pin.
- `upmask` for ports that permit pullups to be specified on a pin basis. This
mask indicates what pins should have pullups activated. A 1 indicates the pullups is on.

**downmask** for ports that permit pulldowns to be specified on a pin basis. This mask indicates what pins should have pulldowns activated. A 1 indicates the pulldowns is on.

**Returns:** undefined

**Function:** Sets the input pullups. TRUE will activate, and a FALSE will deactivate.

**Availability:** Only 14 and 16 bit devices (PCM and PCH). (Note: use SETUP_COUNTERS on PCB parts).

**Requires:** Nothing

**Examples:**

```c
port_a_pullups(FALSE);
```

**Example Files:**

- ex_lcdkb.c
- kbd.c

**Also See:**

- input()
- input_x()
- output_float()

---

**pow( ) pwr( )**

**Syntax:**

- `f = pow (x, y)`
- `f = pwr (x, y)`

**Parameters:**

- `x` and `y` are any float type

**Returns:** A float with precision equal to function parameters `x` and `y`.

**Function:** Calculates X to the Y power.

Note on error handling:
If "errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Range error occurs in the following case:

- pow: when the argument X is negative

**Availability:** All Devices
### printf( ) fprintf( )

**Syntax:**

```
printf (string)
or
printf (cstring, values...)
or
printf (fname, cstring, values...)
fprintf (stream, cstring, values...)
```

**Parameters:**

- **String** is a constant string or an array of characters null terminated.

- **C String** is a constant string. Note that format specifiers cannot be used in RAM strings.

- **Values** is a list of variables separated by commas, fname is a function name to be used for outputting (default is putc is none is specified).

- **Stream** is a stream identifier (a constant byte).

**Returns:** undefined

**Function:**

Outputs a string of characters to either the standard RS-232 pins (first two forms) or to a specified function. Formatting is in accordance with the string argument. When variables are used this string must be a constant. The % character is used within the string to indicate a variable value is to be formatted and output. Longs in the printf may be 16 or 32 bit. A %% will output a single %. Formatting rules for the % follows.

See the Expressions > Constants and Trigraph sections of this manual for other escape character that may be part of the string.

If fprintf() is used then the specified stream is used where printf() defaults to STDOUT (the last USE RS232).
Format:
The format takes the generic form %nt. n is optional and may be 1-9 to specify how many characters are to be outputted, or 01-09 to indicate leading zeros, or 1.1 to 9.9 for floating point and %w output. t is the type and may be one of the following:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Character</td>
</tr>
<tr>
<td>s</td>
<td>String or character</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned int</td>
</tr>
<tr>
<td>d</td>
<td>Signed int</td>
</tr>
<tr>
<td>Lu</td>
<td>Long unsigned int</td>
</tr>
<tr>
<td>Ld</td>
<td>Long signed int</td>
</tr>
<tr>
<td>x</td>
<td>Hex int (lower case)</td>
</tr>
<tr>
<td>X</td>
<td>Hex int (upper case)</td>
</tr>
<tr>
<td>Lx</td>
<td>Hex long int (lower case)</td>
</tr>
<tr>
<td>LX</td>
<td>Hex long int (upper case)</td>
</tr>
<tr>
<td>f</td>
<td>Float with truncated decimal</td>
</tr>
<tr>
<td>g</td>
<td>Float with rounded decimal</td>
</tr>
<tr>
<td>e</td>
<td>Float in exponential format</td>
</tr>
<tr>
<td>w</td>
<td>Unsigned int with decimal place inserted. Specify two numbers for n. The first is a total field width. The second is the desired number of decimal places.</td>
</tr>
</tbody>
</table>

Example formats:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Value=0x12</th>
<th>Value=0xfe</th>
</tr>
</thead>
<tbody>
<tr>
<td>%03u</td>
<td>018</td>
<td>254</td>
</tr>
<tr>
<td>%u</td>
<td>18</td>
<td>254</td>
</tr>
<tr>
<td>%2u</td>
<td>18</td>
<td>*</td>
</tr>
<tr>
<td>%5</td>
<td>18</td>
<td>254</td>
</tr>
<tr>
<td>%d</td>
<td>18</td>
<td>-2</td>
</tr>
<tr>
<td>%x</td>
<td>12</td>
<td>fe</td>
</tr>
<tr>
<td>%X</td>
<td>12</td>
<td>FE</td>
</tr>
<tr>
<td>%4X</td>
<td>0012</td>
<td>00FE</td>
</tr>
<tr>
<td>%3.1w</td>
<td>1.8</td>
<td>25.4</td>
</tr>
</tbody>
</table>

* Result is undefined - Assume garbage.

Availability: All Devices

Requires: #USE RS232 (unless fname is used)

Examples:

```c
byte x,y,z;
printf("HiThere");
printf("RTCCValue=>%2x\n\r", get_rtc());
```
Built-in Functions

```c
printf("%2u %X %4X\n", x, y, z);
printf(LCD_PUTC, "n=%u", n);
```

**Example Files:** ex_admm.c, ex_lcdkb.c

**Also See:** atoi(), puts(), putc(), getc() (for a stream example), RS232 I/O Overview

---

### profileout()

**Syntax:**
- `profileout(string);`
- `profileout(string, value);`
- `profileout(value);`

**Parameters:**
- `string` is any constant string, and `value` can be any constant or variable integer. Despite the length of string the user specifies here, the code profile run-time will actually only send a one or two byte identifier tag to the code profile tool to keep transmission and execution time to a minimum.

**Returns:**
- Undefined

**Function:**
- Typically the code profiler will log and display function entry and exits, to show the call sequence and profile the execution time of the functions. By using `profileout()`, the user can add any message or display any variable in the code profile tool. Most messages sent by `profileout()` are displayed in the 'Data Messages' and 'Call Sequence' screens of the code profile tool.
- If a `profileout(string)` is used and the first word of string is "START", the code profile tool will then measure the time it takes until it sees the same `profileout(string)` where the "START" is replaced with "STOP". This measurement is then displayed in the 'Statistics' screen of the code profile tool, using string as the name (without "START" or "STOP")

**Availability:**
- Any device.

**Requires:**
- `#use profile()` used somewhere in the project source code.

**Examples:**
- // send a simple string.
  `profileout("This is a text string");`
- // send a variable with a string identifier.
  `profileout("RemoteSensor=", adc);`
- // just send a variable.
  `profileout(adc);`
- // time how long a block of code takes to execute.
// this will be displayed in the 'Statistics' of the
// Code Profile tool.
profileout("start my algorithm");
/* code goes here */
profileout("stop my algorithm");

Example Files: ex_profile.c
Also See: #use profile(), #profile, Code Profile overview

### psmc_blanking( )

**Syntax:**

```c
psmc_blanking(unit, rising_edge, rise_time, falling_edge, fall_time);
```

**Parameters:**

- **unit** is the PSMC unit number 1-4
- **rising_edge** are the events that are ignored after the signal activates.
- **rise_time** is the time in ticks (0-255) that the above events are ignored.
- **falling_edge** are the events that are ignored after the signal goes inactive.
- **fall_time** is the time in ticks (0-255) that the above events are ignored.

**Events:**

- PSMC_EVENT_C1OUT
- PSMC_EVENT_C2OUT
- PSMC_EVENT_C3OUT
- PSMC_EVENT_C4OUT
- PSMC_EVENT_IN_PIN

**Returns:** undefined

**Function:** This function is used when system noise can cause an incorrect trigger from one of the specified events. This function allows for ignoring these events for a period of time around either edge of the signal. See
setup_psmc() for a definition of a tick.

Pass a 0 or FALSE for the events to disable blanking for an edge.

**Availability:** All devices equipped with PSMC module.

**Requires:**

**Examples:**

<table>
<thead>
<tr>
<th>Example Files:</th>
<th>None</th>
</tr>
</thead>
</table>

**Also See:** setup_psmc(), psmc_deadband(), psmc_sync(), psmc_modulation(), psmc_shutdown(), psmc_duty(), psmc_freq_adjust(), psmc_pins()

---

**psmc_deadband( )**

**Syntax:**

```plaintext
psmc_deadband(unit, rising_edge, falling_edge);
```

**Parameters:**

- `unit` is the PSMC unit number 1-4
- `rising_edge` is the deadband time in ticks after the signal goes active. If this function is not called, 0 is used.
- `falling_edge` is the deadband time in ticks after the signal goes inactive. If this function is not called, 0 is used.

**Returns:** undefined

**Function:**

This function sets the deadband time values. Deadbands are a gap in time where both sides of a complementary signal are forced to be inactive. The time values are in ticks. See setup_psmc() for a definition of a tick.

**Availability:** All devices equipped with PSMC module.
ReQUIRES: undefined

EXAMPLES:  // 5 tick deadband when the signal goes active.
            psmc_deadband(1, 5, 0);

Example Files: None

Also See: setup_psmc(), psmc_sync(), psmc_blanking(), psmc_modulation(),
          psmc_shutdown(), psmc_duty(), psmc_freq_adjust(), psmc_pins()

psmc_duty( )

Syntax:  psmc_pins(unit, pins_used, pins_active_low);

Parameters:  unit is the PSMC unit number 1-4

  fall_time is the time in ticks that the signal goes inactive (after the start
  of the period) assuming the event PSMC_EVENT_TIME has been
  specified in the setup_psmc().

Returns:  Undefined

Function:  This function changes the fall time (within the period) for the active
           signal. This can be used to change the duty of the active pulse. Note
           that the time is NOT a percentage nor is it the time the signal is active. It
           is the time from the start of the period that the signal will go inactive. If
           the rise_time was set to 0, then this time is the total time the signal will
           be active.

Availability:  All devices equipped with PSMC module.
**Built-in Functions**

**Examples:**

// For a 10khz PWM, based on Fosc divided by 1
// the following sets the duty from
// 0% to 100% baed on the ADC reading
while(TRUE) {
    psmc_duty(1,(read_adc()*(int16)10)/25)*
    (getenv("CLOCK")/1000000));
}

Example Files: None

Also See: setup_psmc(), psmc_deadband(), psmc_sync(), psmc_blanking(),
psmc_modulation(), psmc_shutdown(), psmc_freq_adjust(), psmc_pins()

---

### psmc_freq_adjust()

**Syntax:**

```c
psmc_freq_adjust(unit, freq_adjust);
```

**Parameters:**

- `unit` is the PSMC unit number 1-4
- `freq_adjust` is the time in tick/16 increments to add to the period. The value may be 0-15.

**Returns:** Undefined

**Function:** This function adds a fraction of a tick to the period time for some modes of operation.

**Availability:** All devices equipped with PSMC module.

**Requires:**

**Examples:**

Example Files: None

Also See: setup_psmc(), psmc_deadband(), psmc_sync(), psmc_blanking(),
psmc_modulation(), psmc_shutdown(), psmc_dutyt(), psmc_pins()
**psmc_modulation()**

**Syntax:**
```c
psmc_modulation(unit, options);
```

**Parameters:**
- **unit** is the PSMC unit number 1-4
- **Options** may be one of the following:
  - PSMC_MOD_OFF
  - PSMC_MOD_ACTIVE
  - PSMC_MOD_INACTIVE
  - PSMC_MOD_C1OUT
  - PSMC_MOD_C2OUT
  - PSMC_MOD_C3OUT
  - PSMC_MOD_C4OUT
  - PSMC_MOD_CCP1
  - PSMC_MOD_CCP2
  - PSMC_MOD_IN_PIN

The following may be OR'ed with the above:
- PSMC_MOD_INVERT
- PSMC_MOD_NOT_BDF
- PSMC_MOD_NOT_ACE

**Returns:** undefined

**Function:**
This function allows some source to control if the PWM is running or not. The active/inactive are used for software to control the modulation. The other sources are hardware controlled modulation. There are also options to invert the inputs, and to ignore some of the PWM outputs for the purpose of modulation.

**Availability:** All devices equipped with PSMC module.

**Requires:**
### psmc_pins( )

**Syntax:**

```c
psmc_pins(unit, pins_used, pins_active_low);
```

**Parameters:**

- `unit` is the PSMC unit number 1-4
- `used_pins` is the any combination of the following or'ed together:
  - `PSMC_A`
  - `PSMC_B`
  - `PSMC_C`
  - `PSMC_D`
  - `PSMC_E`
  - `PSMC_F`
  - `PSMC_ON_NEXT_PERIOD`

  If the last constant is used, all the changes made take effect on the next period (as opposed to immediate)

- `pins_active_low` is an optional parameter. When used it lists the same pins from above as the pins that should have an inverted polarity.

**Returns:**

- Undefined

**Function:**

This function identified the pins allocated to the PSMC unit, the polarity of those pins and it enables the PSMC unit. The tri-state register for each pin is set to the output state.
Availability: All devices equipped with PSMC module.

Requires:

Examples: // Simple PWM, 10khz out on pin C0 assuming a 20mhz crystal
           // Duty is initially set to 25%
           setup_psmc(1, PSMC)SINGLE,
           PSMC_EVENT_TIME | PSMC_SOURCE_FOSC, us(100,
           PSMC_EVENT_TIME, 0,
           PSMC_EVENT_TIME, us(25));

           psmc_pins(1, PSMC_A);

Example Files: None

Also See: setup_psmc(), psmc_deadbond(), psmc_sync(), psmc_blanking(),
          psmc_modulation(), psmc_shutdown(), psmc_duty(), psmc_freq_adjust()

psmc_shutdown( )

Syntax: psmc_shutdown(unit, options, source, pins_high);
        psmc_shutdown(unit, command);
## Parameters

*unit* is the PSMC unit number 1-4

**Options** may be one of the following:
- PSMC_SHUTDOWN_OFF
- PSMC_SHUTDOWN_NORMAL
- PSMC_SHUTDOWN_AUTO_RESTART

**command** may be one of the following:
- PSMC_SHUTDOWN_RESTART
- PSMC_SHUTDOWN_FORCE
- PSMC_SHUTDOWN_CHECK

**source** may be any of the following or'ed together:
- PSMC_SHUTDOWN_C1OUT
- PSMC_SHUTDOWN_C2OUT
- PSMC_SHUTDOWN_C3OUT
- PSMC_SHUTDOWN_C4OUT
- PSMC_SHUTDOWN_IN_PIN

**pins_high** is any combination of the following or'ed together:
- PSMC_A
- PSMC_B
- PSMC_C
- PSMC_D
- PSMC_E
- PSMC_F

## Returns

Non-zero if the unit is now in shutdown.

## Function

This function implements a shutdown capability. When any of the listed events activate the PSMC unit will shutdown and the output pins are driver low unless they are listed in the pins that will be driven high.

The auto restart option will restart when the condition goes inactive, otherwise a call with the restart command must be used. Software can force a shutdown with the force command.

## Availability

All devices equipped with PSMC module.
psmc_sync()

Syntax: psmc_sync(slave_unit, master_unit, options);

Parameters: slave_unit is the PSMC unit number 1-4 to be controlled.

    master_unit is the PSMC unit number 1-4 to be synchronized to

Options may be:

    • PSMC_SOURCE_IS_PHASE
    • PSMC_SOURCE_IS_PERIOD
    • PSMC_DISCONNECT

The following may be OR'ed with the above:

    • PSMC_INVERT_DUTY
    • PSMC_INVET_PERIOD

Returns: undefined

Function: This function allows one PSMC unit (the slave) to be synchronized (the outputs) with another PSMC unit (the master).

Availability: All devices equipped with PSMC module.
Built-in Functions

Requires:

Examples:

Example Files: None

Also See: setup_psmc(), psmc_deadband(), psmc_sync(), psmc_modulation(), psmc_shutdown(), psmc_duty(), psmc_freq_adjust(), psmc_pins()

**psp_output_full()**  **psp_input_full()**  **psp_overflow()**

**Syntax:**

```c
result = psp_output_full()
result = psp_input_full()
result = psp_overflow()
result = psp_error(); // EPMP only
result = psp_timeout(); // EPMP only
```

**Parameters:** None

**Returns:** A 0 (FALSE) or 1 (TRUE)

**Function:** These functions check the Parallel Slave Port (PSP) for the indicated conditions and return TRUE or FALSE.

**Availability:** This function is only available on devices with PSP hardware on chips.

**Requires:** Nothing

**Examples:**

```c
while ( psp_output_full() )
psp_data = command;
while (! psp_input_full() )
if ( psp_overflow() )
    error = TRUE;
else
    data = psp_data;
```

**Example Files:** ex_psp.c
Also See: setup_psp(), PSP Overview

## psp_read( )

**Syntax:**

```c
Result = psp_read ( );
Result = psp_read ( address );
```

**Parameters:**

- `address`: The address of the buffer location that needs to be read. If address is not specified, use the function psp_read() which will read the next buffer location.

**Returns:**

A byte of data.

**Function:**

psp_read() will read a byte of data from the next buffer location and psp_read( address ) will read the buffer location `address`.

**Availability:**

Only the devices with a built in Parallel Master Port module of Enhanced Parallel Master Port module.

**Requires:**

Nothing.

**Examples:**

```c
Result = psp_read();       // Reads next byte of data
Result = psp_read(3);      // Reads the buffer location 3
```

**Example Files:**

None

**Also See:**

setup_pmp(), pmp_address(), pmp_read(), psp_read(), psp_write(), pmp_write(), pmp_output_full(), psp_input_full(), psp_overflow(), pmp_output_full(), pmp_input_full(), pmp_overflow().

See header file for device selected.

## psp_write( )

**Syntax:**

```c
psp_write ( data );
psp_write( address, data );
```

**Parameters:**

- `address`: The buffer location that needs to be written to
- `data`: The byte of data to be written
Returns: Undefined.

Function: This will write a byte of data to the next buffer location or will write a byte to the specified buffer location.

Availability: Only the devices with a built in Parallel Master Port module or Enhanced Parallel Master Port module.

Requires: Nothing.

Examples: psp_write( data ); // Write the data byte to // the next buffer location.

Example Files: None

Also See: setup_pmp(), pmp_address(), pmp_read(), pmp_write(), pmp_write(), pmp_output_full(), pmp_input_full(), pmp_overflow(), pmp_output_full(), pmp_input_full(), pmp_overflow(). See header file for device selected.

putc_send( ); fputc_send( );

Syntax: putc_send();
              fputc_send(stream);

Parameters: stream – parameter specifying the stream defined in #USE RS232.

Returns: Nothing

Function: Function used to transmit bytes loaded in transmit buffer over RS232. Depending on the options used in #USE RS232 controls if function is available and how it works.

If using hardware UARTx with NOTXISR option it will check if currently transmitting. If not transmitting it will then check for data in transmit buffer. If there is data in transmit buffer it will load next byte from transmit buffer into the hardware TX buffer, unless using CTS flow control option. In that case it will first check to see if CTS line is at its active state before loading next byte from transmit buffer into the hardware TX buffer.

If using hardware UARTx with TXISR option, function only available if using CTS flow control option, it will test to see if the TBEx interrupt is
enabled. If not enabled it will then test for data in transmit buffer to send. If there is data to send it will then test the CTS flow control line and if at its active state it will enable the TBEx interrupt. When using the TXISR mode the TBEx interrupt takes care off moving data from the transmit buffer into the hardware TX buffer.

If using software RS232, only useful if using CTS flow control, it will check if there is data in transmit buffer to send. If there is data it will then check the CTS flow control line, and if at its active state it will clock out the next data byte.

**Availability:** All devices

**Requires:** #USE RS232

**Examples:**

```c
USE_RS232(UART1,BAUD=9600,TRANSMIT_BUFFER=50,NOTXISR)
printf("Testing Transmit Buffer");
while(TRUE){
    putc_send();
}
```

**Example Files:** None

**Also See:** USE_RS232(), RCV_BUFFER_FULL(), TX_BUFFER_FULL(), TX_BUFFER_BYTES(), GET(), PUTC(), RINTF(), SETUP_UART(), PUTC(), SEND

### pwm_off()

**Syntax:**

```c
pwm_off([stream]);
```

**Parameters:**

- **stream** – optional parameter specifying the stream defined in #USE PWM.

**Returns:**

Nothing.

**Function:**

To turn off the PWM signal.

**Availability:**

All devices.

**Requires:** #USE PWM
### Examples:

```c
#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
while(TRUE){
    if(kbhit()){
        c = getc();

        if(c=='F')
            pwm_off();
    }
}
```

### Example Files:

None

### Also See:

- `#use_pwm`
- `pwm_on()`
- `pwm_set_duty_percent()`
- `pwm_set_duty()`
- `pwm_set_frequency()`

---

## pwm_on()

### Syntax:

`pwm_on([stream]);`

### Parameters:

- `stream` – optional parameter specifying the stream defined in `#USE PWM`.

### Returns:

Nothing.

### Function:

To turn on the PWM signal.

### Availability:

All devices.

### Requires:

`#USE PWM`

### Examples:

```c
#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
while(TRUE){
    if(kbhit()){
        c = getc();

        if(c=='O')
            pwm_on();
    }
}
```

### Example Files:

None
Also See:  
#use_pwm, pwm_off(), pwm_set_duty_percent(), pwm_set_duty(), pwm_set_frequency()

## pwm_set_duty()

### Syntax:
```
pwm_set_duty([stream], duty);
```

### Parameters:
- **stream** – optional parameter specifying the stream defined in #USE PWM.
- **duty** – an int16 constant or variable specifying the new PWM high time.

### Returns:
Nothing.

### Function:
To change the duty cycle of the PWM signal. The duty cycle percentage depends on the period of the PWM signal. This function is faster than pwm_set_duty_percent(), but requires you to know what the period of the PWM signal is.

### Availability:
All devices.

### Requires:
#USE PWM

### Examples:
```
#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
```

### Example Files:
None

### Also See:
#use_pwm, pwm_on(), pwm_off(), pwm_set_frequency(), pwm_set_duty_percent()

## pwm_set_duty_percent

### Syntax:
```
pwm_set_duty_percent([stream]), percent
```

### Parameters:
- **stream** – optional parameter specifying the stream defined in #USE PWM.
- **percent** – an int16 constant or variable ranging from 0 to 1000 specifying the new PWM duty cycle, D is 0% and 1000 is 100.0%.

### Returns:
Nothing.

### Function:
To change the duty cycle of the PWM signal. The duty cycle percentage depends on the period of the PWM signal. This function is faster than pwm_set_duty(), but requires you to know what the period of the PWM signal is.
## Built-in Functions

<table>
<thead>
<tr>
<th>Function:</th>
<th>To change the duty cycle of the PWM signal. Duty cycle percentage is based off the current frequency/period of the PWM signal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability:</td>
<td>All devices.</td>
</tr>
<tr>
<td>Requires:</td>
<td>#USE PWM</td>
</tr>
<tr>
<td>Examples:</td>
<td></td>
</tr>
</tbody>
</table>

```cpp
#USE_PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
pwm_set_duty_percent(500); // set PWM duty cycle to 50%
```

### Example Files:
None

### Also See:

- #use_pwm
- pwm_on()
- pwm_off()
- pwm_set_frequency()
- pwm_set_duty()

---

### pwm_set_frequency

<table>
<thead>
<tr>
<th>Syntax:</th>
<th><code>pwm_set_frequency([stream], frequency);</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters:</td>
<td></td>
</tr>
</tbody>
</table>

- **stream** – optional parameter specifying the stream defined in #USE PWM.
- **frequency** – an int32 constant or variable specifying the new PWM frequency.

| Returns: | Nothing. |
| Function: | To change the frequency of the PWM signal. Warning this may change the resolution of the PWM signal. |
| Availability: | All devices. |
| Requires: | #USE PWM |
| Examples: | 

```cpp
#USE_PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
pwm_set_frequency(1000); // set PWM frequency to 1kHz
```

### Example Files:
None

### Also See:

- #use_pwm
- pwm_on()
- pwm_off()
- pwm_set_frequency_percent
- pwm_set_duty()
pwm1_interrupt_active( )
pwm2_interrupt_active( )
pwm3_interrupt_active( )
pwm4_interrupt_active( )
pwm5_interrupt_active( )
pwm6_interrupt_active( )

Syntax:

result_pwm1_interrupt_active (interrupt)
result_pwm2_interrupt_active (interrupt)
result_pwm3_interrupt_active (interrupt)
result_pwm4_interrupt_active (interrupt)
result_pwm5_interrupt_active (interrupt)
result_pwm6_interrupt_active (interrupt)

Parameters:

interrupt - 8-bit constant or variable. Constants are defined in the device's header file as:

- PWM_PERIOD_INTERRUPT
- PWM_DUTY_INTERRUPT
- PWM_PHASE_INTERRUPT
- PWM_OFFSET_INTERRUPT

Returns:

TRUE if interrupt is active. FALSE if interrupt is not active.

Function:

Tests to see if one of the above PWM interrupts is active, interrupt flag is set.

Availability:

Devices with a 16-bit PWM module.

Requires:

Nothing

Examples:

if(pwm1_interrupt_active(PWM_PERIOD_INTERRUPT))
clear_pwm1_interrupt(PWM_PERIOD_INTERRUPT);

Also See:

setup_pwm(), set_pwm_duty(), set_pwm_phase(), set_pwm_period(),
set_pwm_offset(), enable_pwm_interrupt(), clear_pwm_interrupt(),
disable_pwm_interrupt()
### qei_get_count()

**Syntax:**  
```c
value = qei_get_count([unit]);
```

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value</code></td>
<td>The 16-bit value of the position counter.</td>
</tr>
<tr>
<td><code>unit</code></td>
<td>Optional unit number, defaults to 1.</td>
</tr>
</tbody>
</table>

**Returns:**  
`void`

**Function:**  
Reads the current 16-bit value of the position counter.

**Availability:**  
Devices that have the QEI module.

**Requires:**  
Nothing.

**Examples:**  
```c
value = qei_get_counter();
```

**Example Files:**  
None

**Also See:**  
`setup_qei()`, `qei_set_count()`, `qei_status()`.

### qei_set_count()

**Syntax:**  
```c
qei_set_count([unit], value);
```

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value</code></td>
<td>The 16-bit value of the position counter.</td>
</tr>
<tr>
<td><code>unit</code></td>
<td>Optional unit number, defaults to 1.</td>
</tr>
</tbody>
</table>

**Returns:**  
`void`

**Function:**  
Write a 16-bit value to the position counter.

**Availability:**  
Devices that have the QEI module.

**Requires:**  
Nothing.

**Examples:**  
```c
qei_set_counter(value);
```

**Example Files:**  
None

**Also See:**  
`setup_qei()`, `qei_get_count()`, `qei_status()`.
### qei_status()

**Syntax:**

```c
status = qei_status([unit]);
```

**Parameters:**

- `status`: The status of the QEI module
- `unit`: Optional unit number, defaults to 1.

**Returns:**

`void`

**Function:**

Returns the status of the QEI module.

**Availability:**

Devices that have the QEI module.

**Requires:**

Nothing.

**Examples:**

```c
status = qei_status();
```

**Example Files:** None

**Also See:** `setup_qei()`, `qei_set_count()`, `qei_get_count()`.

### qsort()

**Syntax:**

```c
qsort(base, num, width, compare)
```

**Parameters:**

- `base`: Pointer to array of sort data
- `num`: Number of elements
- `width`: Width of elements
- `compare`: Function that compares two elements

**Returns:**

`None`

**Function:**

Performs the shell-metzner sort (not the quick sort algorithm). The contents of the array are sorted into ascending order according to a comparison function pointed to by compare.

**Availability:**

All devices

**Requires:**

```c
#include <stdlib.h>
```

**Examples:**

```c
int nums[5]={ 2,3,1,5,4};
```
```c
int compar(void *arg1, void *arg2);

void main() {
    qsort (nums, 5, sizeof(int), compar);
}

int compar(void *arg1, void *arg2) {
    if (* (int *) arg1 < ( * (int *) arg2) return -1
    else if ( * (int *) arg1 == ( * (int *) arg2) return 0
    else return 1;
}
```

### Example Files:

```
ex_qsort.c
```

### Also See:

```
bsearch()
```

---

### `rand( )`

**Syntax:**

```
re=rand()
```

**Parameters:**

None

**Returns:**

A pseudo-random integer.

**Function:**

The `rand` function returns a sequence of pseudo-random integers in the range of 0 to RAND_MAX.

**Availability:**

All devices

**Requires:**

```
#include <stdlib.h>
```

**Examples:**

```
int I;
I=rand();
```
### rcv_buffer_bytes()

**Syntax:**

```cpp
value = rcv_buffer_bytes([stream]);
```

**Parameters:**

- `stream` – optional parameter specifying the stream defined in `#USE RS232`.

**Returns:**

Number of bytes in receive buffer that still need to be retrieved.

**Function:**

Function to determine the number of bytes in receive buffer that still need to be retrieved.

**Availability:**

All devices

**Requires:**

`#USE RS232`

**Examples:**

```cpp
#USE_RS232(UART1, BAUD=9600, RECEIVE_BUFFER=100)
void main(void) {
    char c;
    if(rcv_buffer_bytes() > 10)
        c = getc();
}
```

**Example Files:**

None

**Also See:**

`rand()`, `rcv_buffer_bytes()`, `GETC()`, `PUTC()`, `PRINTF()`, `SETUP_UART()`, `putc_send()`
rcv_buffer_full( )

Syntax: 
value = rcv_buffer_full([stream]);

Parameters: 
stream – optional parameter specifying the stream defined in #USE RS232.

Returns: 
TRUE if receive buffer is full, FALSE otherwise.

Function: 
Function to test if the receive buffer is full.

Availability: 
All devices

Requires: 
#USE RS232

Examples: 
#USE_RS232(UART1,BAUD=9600,RECEIVE_BUFFER=100)
void main(void) {
    char c;
    if(rcv_buffer_full())
        c = getc();
}

Example Files: 
None

Also See:  USE_RS232(), RCV_BUFFER_BYTES(), TX_BUFFER_BYTES(), 
          TX_BUFFER_FULL(), GETC(), PUTC(), PRINTF(), SETUP_UART(), 
          PUTC_SEND()

read_adc( )       read_adc2( )

Syntax: 
value = read_adc (mode)
value = read_adc2 ([mode])
value = read_adc (mode,[channel]) // dsPIC33EPxxGSxxx family only

Parameters: 
mode is an optional parameter. If used the values may be:
ADC_START_AND_READ (continually takes readings, this is the default)
ADC_START_ONLY (starts the conversion and returns)
ADC_READ_ONLY (reads last conversion result)
channel is an optional parameter for specifying the channel to start the conversion on and/or read the result from. If not specified will use channel specified in last call to set_adc_channel(), read_adc(), or adc_done(). Only available for dsPIC33EPxxGSxxx family.

Returns: Either a 8 or 16 bit int depending on #DEVICE ADC= directive.

Function: This function will read the digital value from the analog to digital converter. Calls to setup_adc(), setup_adc_ports() and set_adc_channel() should be made sometime before this function is called. The range of the return value depends on number of bits in the chips A/D converter and the setting in the #DEVICE ADC= directive as follows:

<table>
<thead>
<tr>
<th>#DEVICE</th>
<th>10 bit</th>
<th>12 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC=8</td>
<td>00-FF</td>
<td>00-FF</td>
</tr>
<tr>
<td>ADC=10</td>
<td>0-3FF</td>
<td>0-3FF</td>
</tr>
<tr>
<td>ADC=11</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ADC=12</td>
<td>0-FFC</td>
<td>0-FFF</td>
</tr>
<tr>
<td>ADC=16</td>
<td>0-FFC0</td>
<td>0-FF0</td>
</tr>
</tbody>
</table>

Note: x is not defined

Availability: Only available on devices with built in analog to digital converters.

Requires: Pin constants are defined in the devices .h file.

Examples: int16 value;
set_up_adc_ports(sAN0|sAN1, VSS_VDD);
set_up_adc(ADC_CLOCK_DIV_4|ADC_TAD_MUL_8);

while (TRUE)
{
    set_adc_channel(0);
    value = read_adc();
    printf("Pin AN0 A/C value = %LX
n", value);

delay_ms(5000);

    set_adc_channel(1);
    read_adc(ADC_START_ONLY);
    ...
    value = read_adc(ADC_READ_ONLY);
    printf("Pin AN1 A/D value = %LX
n", value);
}

Example Files: ex_admm.c,
**Built-in Functions**

### read_configuration_memory( )

**Syntax:**

```c
read_configuration_memory([offset], ramPtr, n)
```

**Parameters:**

- `ramPtr` is the destination pointer for the read results
- `count` is an 8 bit integer
- `offset` is an optional parameter specifying the offset into configuration memory to start reading from, offset defaults to zero if not used.

**Returns:**

undefined

**Function:**

Reads `n` bytes of configuration memory and saves the values to `ramPtr`.

**Availability:**

All

**Requires:**

Nothing

**Examples:**

```c
int data[6];
read_configuration_memory(data,6);
```

**Example Files:**

None

**Also See:**

write_configuration_memory(), read_program_memory(), Configuration Memory Overview

### read_eeprom( )

**Syntax:**

```c
value = read_eeprom(address, [N])
read_eeprom(address, variable)
read_eeprom(address, pointer, N)
```

**Parameters:**

- `address` is an 8 bit or 16 bit int depending on the part
- `N` specifies the number of EEPROM bytes to read
- `variable` is a specified location to store EEPROM read results
- `pointer` is a pointer to location to store EEPROM read results

**Returns:**

An 16 bit int

**Function:**

By default the function reads a word from EEPROM at the specified address. The number of bytes to read can optionally be defined by argument `N`. If a variable is used as an argument, then EEPROM is read...
and the results are placed in the variable until the variable data size is full. Finally, if a pointer is used as an argument, then n bytes of EEPROM at the given address are read to the pointer.

**Availability:** This command is only for parts with built-in EEPROMS

**Requires:** Nothing

**Examples:**
```c
#define LAST_VOLUME 10
volume = read_EEPROM (LAST_VOLUME);
```

**Example Files:** None

**Also See:** write_eeprom(), Data Eeprom Overview

---

### read_extended_ram()

**Syntax:**
```c
read_extended_ram(page,address,data,count);
```

**Parameters:**
- **page** – the page in extended RAM to read from
- **address** – the address on the selected page to start reading from
- **data** – pointer to the variable to return the data to
- **count** – the number of bytes to read (0-32768)

**Returns:** Undefined

**Function:** To read data from the extended RAM of the PIC.

**Availability:** On devices with more then 30K of RAM.

**Requires:** Nothing

**Examples:**
```c
unsigned int8 data[8];
read_extended_ram(1,0x0000,data,8);
```

**Example Files:** None

**Also See:** read_extended_ram(), Extended RAM Overview

---

338
### read_program_memory( )

**Syntax:**

```plaintext
READ_PROGRAM_MEMORY (address, dataptr, count);
```

**Parameters:**

- `address` is 32 bits. The least significant bit should always be 0 in PCM.
- `dataptr` is a pointer to one or more bytes.
- `count` is a 16 bit integer on PIC16 and 16-bit for PIC18

**Returns:** undefined

**Function:** Reads `count` bytes from program memory at `address` to RAM at `dataptr`. Due to the 24-bit program instruction size on the PCD devices, every fourth byte will be read as 0x00

**Availability:** Only devices that allow reads from program memory.

**Requires:** Nothing

**Examples:**

```plaintext
char buffer[64];
read_external_memory(0x40000, buffer, 64);
```

**Example Files:** None

**Also See:** `write_program_memory()`, Program Eeprom Overview

---

### read_high_speed_adc( )

**Syntax:**

```plaintext
read_high_speed_adc(pair, mode, result); // Individual start and
read or

// read only
read_high_speed_adc(pair, result); // Individual start and
read
read_high_speed_adc(pair); // Individual start
only
read_high_speed_adc(mode, result); // Global start and read
or

read_high_speed_adc(result); // Global start and
read
```

---

339
read_high_speed_adc();  // Global start only

Parameters:

- **pair** – Optional parameter that determines which ADC pair number to start and/or read. Valid values are 0 to total number of ADC pairs. 0 starts and/or reads ADC pair AN0 and AN1, 1 starts and/or reads ADC pair AN2 and AN3, etc. If omitted then a global start and/or read will be performed.

- **mode** – Optional parameter, if used the values may be:
  - ADC_START_AND_READ (starts conversion and reads result)
  - ADC_START_ONLY (starts conversion and returns)
  - ADC_READ_ONLY (reads conversion result)

- **result** – Pointer to return ADC conversion too. Parameter is optional, if not used the read_fast_adc() function can only perform a start.

Returns: Undefined

Function:

This function is used to start an analog to digital conversion and/or read the digital value when the conversion is complete. Calls to setup_high_speed_adc() and setup_high_speed_adc_pairs() should be made sometime before this function is called.

When using this function to perform an individual start and read or individual start only, the function assumes that the pair's trigger source was set to INDIVIDUALSOFTWARE_TRIGGER.

When using this function to perform a global start and read, global start only, or global read only. The function will perform the following steps:

1. Determine which ADC pairs are set for GLOBALSOFTWARE_TRIGGER.
2. Clear the corresponding ready flags (if doing a start).
3. Set the global software trigger (if doing a start).
4. Read the corresponding ADC pairs in order from lowest to highest (if doing a read).
5. Clear the corresponding ready flags (if
function to perform an individual read only. The function can read the ADC result from any trigger source.

**Availability:** Only on dsPIC33FJxxGSxxx devices.

**Requires:** Constants are define in the device .h file.

**Examples:**

```c
//Individual start and read
int16 result[2];

setup_high_speed_adc(ADC_CLOCK_DIV_4);
setup_high_speed_adc_pair(0, INDIVIDUAL_SOFTWARE_TRIGGER);
read_high_speed_adc(0, result); //starts conversion for AN0 and AN1 and stores
                                //result in result[0] and result[1]

//Global start and read
int16 result[4];

setup_high_speed_adc(ADC_CLOCK_DIV_4);
setup_high_speed_adc_pair(0, GLOBAL_SOFTWARE_TRIGGER);
setup_high_speed_adc_pair(4, GLOBAL_SOFTWARE_TRIGGER);
read_high_speed_adc(result); //starts conversion for AN0, AN1,
                             //AN8 and AN9 and
                             //stores result in result[0],
                             //stores result in result[0],
result //[result[1], result[2]
                                and result[3]
```

**Example Files:**
None

**Also See:**
setup_high_speed_adc(), setup_high_speed_adc_pair(),
high_speed_adc_done()

---

**read_rom_memory()**

**Syntax:**  
READ_ROM_MEMORY (address, dataptr, count);

**Parameters:**
- `address` is 32 bits. The least significant bit should always be 0.
- `dataptr` is a pointer to one or more bytes.
- `count` is a 16 bit integer
Returns: undefined

Function: Reads count bytes from program memory at address to dataptr. Due to the 24 bit program instruction size on the PCD devices, three bytes are read from each address location.

Availability: Only devices that allow reads from program memory.

Requires: Nothing

Examples: char buffer[64];
read_program_memory(0x40000, buffer, 64);

Example Files: None

Also See: write_eeprom(), read_eeprom(), Program eeprom overview

### read_sd_adc()

Syntax: value = read_sd_adc();

Parameters: None

Returns: A signed 32 bit int.

Function: To poll the SDRDY bit and if set return the signed 32 bit value stored in the SD1RESH and SD1RESL registers, and clear the SDRDY bit. The result returned depends on settings made with the setup_sd_adc() function, but will always be a signed int32 value with the most significant bits being meaningful. Refer to Section 66, 16-bit Sigma-Delta A/D Converter, of the PIC24F Family Reference Manual for more information on the module and the result format.

Availability: Only devices with a Sigma-Delta Analog to Digital Converter (SD ADC) module.

Examples: value = read_sd_adc()

Example Files: None
realloc( )

**Syntax:**

```c
realloc (ptr, size)
```

**Parameters:**

- `ptr` is a null pointer or a pointer previously returned by `calloc` or `malloc` or `realloc` function, size is an integer representing the number of bytes to be allocated.

**Returns:**

A pointer to the possibly moved allocated memory, if any. Returns null otherwise.

**Function:**

The `realloc` function changes the size of the object pointed to by the `ptr` to the size specified by the `size`. The contents of the object shall be unchanged up to the lesser of new and old sizes. If the new size is larger, the value of the newly allocated space is indeterminate. If `ptr` is a null pointer, the `realloc` function behaves like `malloc` function for the specified `size`. If the `ptr` does not match a pointer earlier returned by the `calloc`, `malloc` or `realloc`, or if the space has been deallocated by a call to `free` or `realloc` function, the behavior is undefined. If the space cannot be allocated, the object pointed to by `ptr` is unchanged. If `size` is zero and the `ptr` is not a null pointer, the object is to be freed.

**Availability:**

All devices

**Requires:**

```c
#include <stdlib.h>
```

**Examples:**

```c
int * iptr;
iptr=malloc(10);
realloc(iptr,20)
// iptr will point to a block of memory of 20 bytes, if available.
```

**Example Files:**

None

**Also See:**

`malloc()`, `free()`, `calloc()`
# release_io()

**Syntax:**

release_io();

**Parameters:**

none

**Returns:**

nothing

**Function:**

The function releases the I/O pins after the device wakes up from deep sleep, allowing the state of the I/O pins to change.

**Availability:**

Devices with a deep sleep module.

**Requires:**

Nothing

**Examples:**

```c
unsigned int16 restart;
restart = restart_cause();
if(restart == RTC_FROM_DS)
    release_io();
```

**Also See:**

sleep()

---

# reset_cpu()

**Syntax:**

reset_cpu()

**Parameters:**

None

**Returns:**

This function never returns

**Function:**

This is a general purpose device reset. It will jump to location 0 on PCB and PCM parts and also reset the registers to power-up state on the PIC18XXX.

**Availability:**

All devices

**Requires:**

Nothing

**Examples:**

```c
if(checksum!=0)
    reset_cpu();
```
### Example Files:

| None |

### Also See:

| None |

## `restart_cause()`

### Syntax:

```
value = restart_cause()
```

### Parameters:

None

### Returns:

A value indicating the cause of the last processor reset. The actual values are device dependent. See the device .h file for specific values for a specific device. Some example values are: `RESTART_POWER_UP`, `RESTART_BROWNOUT`, `RESTART_WDT` and `RESTART_MCLR`.

### Function:

Returns the cause of the last processor reset.

In order for the result to be accurate, it should be called immediately in `main()`.

### Availability:

All devices

### Requires:

Constants are defined in the devices .h file.

### Examples:

```c
switch ( restart_cause() ) {
    case RESTART_BROWNOUT:
    case RESTART_WDT:
    case RESTART_MCLR:
        handle_error();
}
```

### Example Files:

| `ex_wdt.c` |

### Also See:

| `restart_wdt()`, `reset_cpu()` |
**restart_wdt( )**

**Syntax:**

restart_wdt()

**Parameters:**

None

**Returns:**

undefined

**Function:**

Restarts the watchdog timer. If the watchdog timer is enabled, this must be called periodically to prevent the processor from resetting.

The watchdog timer is used to cause a hardware reset if the software appears to be stuck.

The timer must be enabled, the timeout time set and software must periodically restart the timer. These are done differently on the PCB/PCM and PCH parts as follows:

<table>
<thead>
<tr>
<th>Enable/Disable</th>
<th>PCB/PCM</th>
<th>PCH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>setup_wdt()</td>
<td>#fuses</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timeout time</th>
<th>PCB/PCM</th>
<th>PCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup_wdt()</td>
<td></td>
<td>#fuses</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>restart</th>
<th>PCB/PCM</th>
<th>PCH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>restart_wdt()</td>
<td>restart_wdt()</td>
</tr>
</tbody>
</table>

**Availability:**

All devices

**Requires:**

#FUSES

**Examples:**

```c
#define WDT // PCB/PCM example

// See setup_wdt for a
// PIC18 example

main() {
  setup_wdt(WDT_2304MS);
  while (TRUE) {
    restart_wdt();
    perform_activity();
  }
}
```

**Example Files:**

ex_wdt.c

**Also See:**

#FUSES, setup_wdt(), WDT or Watch Dog Timer Overview
rotate_left( )

Syntax: \[ \text{rotate\_left (address, bytes)} \]

Parameters:  
- **address** is a pointer to memory  
- **bytes** is a count of the number of bytes to work with.

Returns: undefined

Function: Rotates a bit through an array or structure. The address may be an array identifier or an address to a byte or structure (such as &data). Bit 0 of the lowest BYTE in RAM is considered the LSB.

Availability: All devices

Requires: Nothing

Examples:  
\[
x = 0x86;  
\text{rotate\_left( } &x, 1);  
// x is now 0x0d
\]

Example Files: None

Also See: rotate_right(), shift_left(), shift_right()

rotate_right( )

Syntax: \[ \text{rotate\_right (address, bytes)} \]

Parameters:  
- **address** is a pointer to memory,  
- **bytes** is a count of the number of bytes to work with.

Returns: undefined

Function: Rotates a bit through an array or structure. The address may be an array identifier or an address to a byte or structure (such as &data). Bit 0 of the lowest BYTE in RAM is considered the LSB.

Availability: All devices

Requires: Nothing
Examples:

```c
struct {
    int cell_1 : 4;
    int cell_2 : 4;
    int cell_3 : 4;
    int cell_4 : 4; } cells;
rotate_right( &cells, 2);
rotate_right( &cells, 2);
rotate_right( &cells, 2);
rotate_right( &cells, 2);
// cell_1-->4, 2-->1, 3-->2 and 4--> 3
```

Example Files: None

Also See: rotate_left(), shift_left(), shift_right()

---

**rtc_alarm_read()**

Syntax: `rtc_alarm_read(&datetime);`

Parameters: `datetime` - A structure that will contain the values to be written to the alarm in the RTCC module.

Structure used in read and write functions are defined in the device header file as rtc_time_t

Returns: `void`  

Function: Reads the date and time from the alarm in the RTCC module to structure **datetime**.

Availability: Devices that have the RTCC module.

Requires: Nothing.

Examples: `rtc_alarm_read(&datetime);`

Example Files: None

Also See: rtc_read(), rtc_alarm_read(), rtc_alarm_write(), setup_RTC_alarm(), rtc_write(), setup_rtc()
## rtc_alarm_write( )

**Syntax:**
```c
rtc_alarm_write(&datetime);
```

**Parameters:**
- `datetime`: A structure that will contain the values to be written to the alarm in the RTCC module.

Structure used in read and write functions are defined in the device header file as `rtc_time_t`.

**Returns:**
- `void`

**Function:**
Writes the date and time to the alarm in the RTCC module as specified in the `datetime` structure.

**Availability:**
Devices that have the RTCC module.

**Requires:**
Nothing.

**Examples:**
```c
rtc_alarm_write(&datetime);
```

**Example Files:**
None

**Also See:**
`rtc_read()`, `rtc_alarm_read()`, `rtc_alarm_write()`, `setup_rtc_alarm()`, `rtc_write()`, `setup_rtc`

## rtc_read( )

**Syntax:**
```c
rtc_read(&datetime);
```

**Parameters:**
- `datetime`: A structure that will contain the values returned by the RTCC module.

Structure used in read and write functions are defined in the device header file as `rtc_time_t`.

**Returns:**
- `void`

**Function:**
Reads the current value of Time and Date from the RTCC module and stores the `datetime` structure date time.
### rtc_read(

**Availability:** Devices that have the RTCC module.

**Requires:** Nothing.

**Examples:**
```
rtc_read(&datetime);
```

**Example Files:** `ex_rtcc.c`

**Also See:** `rtc_read()`, `rtc_alarm_read()`, `rtc_alarm_write()`, `setup_rtc_alarm()`, `rtc_write()`, `setup_rtc()`

---

### rtc_write( )

**Syntax:**
```
rtc_write(&datetime);
```

**Parameters:**

- `datetime`: A structure that will contain the values to be written to the RTCC module.

  Structure used in read and write functions are defined in the device header file as `rtc_time_t`.

**Returns:** void

**Function:** Writes the date and time to the RTCC module as specified in the structure `datetime`.

**Availability:** Devices that have the RTCC module.

**Requires:** Nothing.

**Examples:**
```
rtc_write(&datetime);
```

**Example Files:** `ex_rtcc.c`

**Also See:** `rtc_read()`, `rtc_alarm_read()`, `rtc_alarm_write()`, `setup_rtc_alarm()`, `rtc_write()`, `setup_rtc()`
## rtosAwait( )

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

**Syntax:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtosAwait(expre)</td>
<td></td>
</tr>
</tbody>
</table>

**Parameters:**

- `expre` is a logical expression.

**Returns:** None

**Function:**

This function can only be used in an RTOS task. This function waits for `expre` to be true before continuing execution of the rest of the code of the RTOS task. This function allows other tasks to execute while the task waits for `expre` to be true.

**Availability:** All devices

**Requires:** #USE RTOS

**Examples:**

rtosAwait(kbhit());

**Also See:** None

## rtosDisable( )

The RTOS is only included in the PCW, PCWH, and PCWHD software packages.

**Syntax:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtosDisable(task)</td>
<td></td>
</tr>
</tbody>
</table>

**Parameters:**

- `task` is the identifier of a function that is being used as an RTOS task.

**Returns:** None

**Function:**

This function disables a task which causes the task to not execute until enabled by rtosEnable(). All tasks are enabled by default.

**Availability:** All devices

**Requires:** #USE RTOS

**Examples:**

rtosDisable(toggle_green)
rtos_enable( )

The RTOS is only included in the PCW, PCWH, and PCWHD software packages.

Syntax: rtos_enable (task)

Parameters: task is the identifier of a function that is being used as an RTOS task.

Returns: None

Function: This function enables a task to execute at it's specified rate.

Availability: All devices

Requires: #USE RTOS

Examples: rtos_enable(toggle_green);

Also See: rtos_disable()
Built-in Functions

**rtos_msg_read()**

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

**Syntax:**

\[ b = \text{rtos\_msg\_read()} \]

**Parameters:** None

**Returns:** A byte that is a message for the task.

**Function:** This function can only be used inside an RTOS task. This function reads in the next (message) of the queue for the task that the \text{rtos\_msg\_read()} function is used in.

**Availability:** All devices

**Requires:** #USE RTOS

**Examples:**

```
if(rtos_msg_poll()) {
    b = rtos_msg_read();
}
```

**Also See:** rtos msg send(), rtos msg read()

**rtos_msg_send()**

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

**Syntax:**

\[ \text{rtos\_msg\_send}(\text{task}, \text{byte}) \]

**Parameters:**

- **task** is the identifier of a function that is being used as an RTOS task
- **byte** is the byte to send to **task** as a message.
Returns: None

Function: This function can be used anytime after rtos_run() has been called. This function sends a byte long message (byte) to the task identified by \textit{task}.

Availability: All devices

Requires: #USE RTOS

Examples:

\begin{verbatim}
if(kbhit())
{
    rtos_msg_send(echo, getc());
}
\end{verbatim}

Also See: rtos_msg_poll(), rtos_msg_read()

\subsection*{rtos_overrun( )}

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax: \texttt{rtos_overrun([task])}

Parameters: \textit{task} is an optional parameter that is the identifier of a function that is being used as an RTOS task

Returns: A 0 (FALSE) or 1 (TRUE)

Function: This function returns TRUE if the specified task took more time to execute than it was allocated. If no task was specified, then it returns TRUE if any task ran over it's alloted execution time.

Availability: All devices

Requires: #USE RTOS(statistics)

Examples: rtos_overrun()

Also See: None
### rtos_run( )

The RTOS is only included in the PCW, PCWH, and PCWHD software packages.

**Syntax:**  
`rtos_run()`

**Parameters:**  
None

**Returns:**  
None

**Function:**  
This function begins the execution of all enabled RTOS tasks. This function controls the execution of the RTOS tasks at the allocated rate for each task. This function will return only when `rtos_terminate()` is called.

**Availability:**  
All devices

**Requires:**  
`#USE RTOS`

**Examples:**  
`rtos_run()`

**Also See:**  
`rtos_terminate()`

### rtos_signal( )

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

**Syntax:**  
`rtos_signal (sem)`

**Parameters:**  
`sem` is a global variable that represents the current availability of a shared system resource (a semaphore).

**Returns:**  
None

**Function:**  
This function can only be used by an RTOS task. This function increments `sem` to let waiting tasks know that a shared resource is available for use.

**Availability:**  
All devices
**rtos_stats()**

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

**Syntax:**

```c
rtos_stats(task,&stat)
```

**Parameters:**

- `task` is the identifier of a function that is being used as an RTOS task.
- `stat` is a structure containing the following:
  ```c
  struct rtos_stats_struct {
    unsigned int32 task_total_ticks; //number of ticks the task has used
    unsigned int16 task_min_ticks;  //the minimum number of ticks used
    unsigned int16 task_max_ticks;  //the maximum number of ticks used
    unsigned int16 hns_per_tick;    //us = (ticks*hns_per_tick)/10
  }
  ```

**Returns:** Undefined

**Function:** This function returns the statistic data for a specified `task`.

**Availability:** All devices

**Requires:** #USE RTOS(statistics)

**Examples:**

```c
rtos_stats(echo, &stats)
```

**Also See:** None
**rtos_terminate( )**

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

**Syntax:**

```c
rtos_terminate()
```

**Parameters:** None

**Returns:** None

**Function:** This function ends the execution of all RTOS tasks. The execution of the program will continue with the first line of code after the `rtos_run()` call in the program. (This function causes `rtos_run()` to return.)

**Availability:** All devices

**Requires:** #USE RTOS

**Examples:**

```c
rtos_terminate()
```

**Also See:** `rtos_run()`

---

**rtos_wait( )**

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

**Syntax:**

```c
rtos_wait (sem)
```

**Parameters:** `sem` is a global variable that represents the current availability of a shared system resource (a semaphore).

**Returns:** None

**Function:** This function can only be used by an RTOS task. This function waits for `sem` to be greater than 0 (shared resource is available), then decrements `sem` to claim usage of the shared resource and continues the execution of the rest of the code the RTOS task. This function allows other tasks to execute while the task waits for the shared resource to be available.

**Availability:** All devices
rtos_yield()

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax:  rtos_yield()

Parameters:  None

Returns:  None

Function:  This function can only be used in an RTOS task. This function stops the execution of the current task and returns control of the processor to rtos_run(). When the next task executes, it will start it's execution on the line of code after the rtos_yield().

Availability:  All devices

Requires:  #USE RTOS

Examples:  void yield(void)
{  
    printf("Yielding...\n");  
    rtos_yield();  
    printf("Executing code after yield\r\n");  
}

Also See:  None
### set_adc_channel()  set_adc_channel2()

**Syntax:**

- `set_adc_channel(chan [,neg])`
- `set_adc_channel(chan, [differential])` //dsPIC33EPxxGSxxx only
- `set_adc_channel2(chan)`

**Parameters:**

- `chan` is the channel number to select. Channel numbers start at 0 and are labeled in the data sheet AN0, AN1. For devices with a differential ADC it sets the positive channel to use.

- `neg` is optional and is used for devices with a differential ADC only. It sets the negative channel to use, channel numbers can be 0 to 6 or VSS. If no parameter is used the negative channel will be set to VSS by default.

**Returns:**

- `undefined`
- `differential` is an optional parameter to specify if channel is differential or single-ended. TRUE is differential and FALSE is single-ended. Only available for dsPIC3EPxxGSxxx family.

**Function:**

Specifies the channel to use for the next `read_adc()` call. Be aware that you must wait a short time after changing the channel before you can get a valid read. The time varies depending on the impedance of the input source. In general 10us is good for most applications. You need not change the channel before every read if the channel does not change.

**Availability:**

- Only available on devices with built in analog to digital converters

**Requires:**

- Nothing

**Examples:**

- `set_adc_channel(2);`
- `value = read_adc();`

**Example Files:**

- `ex_admm.c`

**Also See:**

- `read_adc()`, `setup_adc()`, `setup_adc_ports()`, `ADC Overview`
**set_adc_trigger()**

**Syntax:**

```c
set_adc_trigger (trigger)
```

**Parameters:**

- `trigger` - ADC trigger source. Constants defined in device's header, see the device's .h file for all options. Some typical options include:
  - ADC_TRIGGER_DISABLED
  - ADC_TRIGGER_ADACT_PIN
  - ADC_TRIGGER_TIMER1
  - ADC_TRIGGER_CCP1

**Returns:** undefined

**Function:**

Sets the Auto-Conversion trigger source for the Analog-to-Digital Converter with Computation (ADC2) Module.

**Availability:**

All devices with an ADC2 Module

**Requires:**

Constants defined in the device's .h file

**Examples:**

```c
set_adc_trigger(ADC_TRIGGER_TIMER1);
```

**Also See:**

ADC Overview, setup_adc(), setup_adc_ports(), set_adc_channel(), read_adc(),
#DEVICE, adc_read(), adc_write(), adc_status()

---

**set_analog_pins()**

**Syntax:**

```c
set_analog_pins(pin, pin, pin, ...)
```

**Parameters:**

- `pin` - pin to set as an analog pin. Pins are defined in the device's .h file. The actual value is a bit address. For example, bit 3 of port A at address 5, would have a value of 5*8+3 or 43. This is defined as follows:
  ```
  #define PIN_A3 43
  ```

**Returns:** undefined

**Function:**

To set which pins are analog and digital. Usage of function depends on method device has for setting pins to analog or digital. For devices with ANSELx, x being the port letter, registers the function is used as described above. For all other devices the function works the same as setup_adc_ports() function.
Refer to the `setup_adc_ports()` page for documentation on how to use.

**Availability:** On all devices with an Analog to Digital Converter

**Requires:** Nothing

**Examples:**

```
set_analog_pins(PIN_A0, PIN_A1, PIN_E1, PIN_B0, PIN_B5);
```

**Example Files:**

Also See: `setup_adc_reference()`, `set_adc_channel()`, `read_adc()`, `setup_adc()`, `setup_adc_ports()`, `ADC Overview`

### scanf( )

**Syntax:**

```
scanf(cstring);
scanf(cstring, values...)
fscanf(stream, cstring, values...)
```

**Parameters:**

- `cstring` is a constant string.
- `values` is a list of variables separated by commas.
- `stream` is a stream identifier.

**Returns:**

0 if a failure occurred, otherwise it returns the number of conversion specifiers that were read in, plus the number of constant strings read in.

**Function:**

Reads in a string of characters from the standard RS-232 pins and formats the string according to the format specifiers. The format specifier character (%) used within the string indicates that a conversion specification is to be done and the value is to be saved into the corresponding argument variable. A % will input a single %.

Formatting rules for the format specifier as follows:

If fscanf() is used, then the specified stream is used, where scanf() defaults to STDIN (the last USE RS232).

Format:

The format takes the generic form `%nt`. `n` is an option and may be 1-99.
specifying the field width, the number of characters to be inputted. \( t \) is the type and maybe one of the following:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>c</strong></td>
<td>Matches a sequence of characters of the number specified by the field width (1 if no field width is specified). The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence.</td>
</tr>
<tr>
<td><strong>s</strong></td>
<td>Matches a sequence of non-white space characters. The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence and a terminating null character, which will be added automatically.</td>
</tr>
<tr>
<td><strong>u</strong></td>
<td>Matches an unsigned decimal integer. The corresponding argument shall be a pointer to an unsigned integer.</td>
</tr>
<tr>
<td><strong>Lu</strong></td>
<td>Matches a long unsigned decimal integer. The corresponding argument shall be a pointer to a long unsigned integer.</td>
</tr>
<tr>
<td><strong>d</strong></td>
<td>Matches a signed decimal integer. The corresponding argument shall be a pointer to a signed integer.</td>
</tr>
<tr>
<td><strong>Ld</strong></td>
<td>Matches a long signed decimal integer. The corresponding argument shall be a pointer to a long signed integer.</td>
</tr>
<tr>
<td><strong>o</strong></td>
<td>Matches a signed or unsigned octal integer. The corresponding argument shall be a pointer to a signed or unsigned integer.</td>
</tr>
<tr>
<td><strong>Lo</strong></td>
<td>Matches a long signed or unsigned octal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.</td>
</tr>
<tr>
<td><strong>x or X</strong></td>
<td>Matches a hexadecimal integer. The corresponding argument shall be a pointer to a signed or unsigned integer.</td>
</tr>
<tr>
<td><strong>Lx or LX</strong></td>
<td>Matches a long hexadecimal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.</td>
</tr>
</tbody>
</table>
### Built-in Functions

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Matches a signed or unsigned integer. The corresponding argument shall be a pointer to a signed or unsigned integer.</td>
</tr>
<tr>
<td>Li</td>
<td>Matches a long signed or unsigned integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.</td>
</tr>
<tr>
<td>f,g or e</td>
<td>Matches a floating point number in decimal or exponential format. The corresponding argument shall be a pointer to a float.</td>
</tr>
<tr>
<td>[</td>
<td>Matches a non-empty sequence of characters from a set of expected characters. The sequence of characters included in the set are made up of all character following the left bracket ([) up to the matching right bracket (]). Unless the first character after the left bracket is a <code>^</code>, in which case the set of characters contain all characters that do not appear between the brackets. If a <code>^</code> character is in the set and is not the first or second, where the first is a <code>^</code>, nor the last character, then the set includes all characters from the character before the <code>-</code> to the character after the <code>-</code>. For example, <code>%[a-z]</code> would include all characters from <code>a</code> to <code>z</code> in the set and <code>%[^a-z]</code> would exclude all characters from <code>a</code> to <code>z</code> from the set. The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence and a terminating null character, which will be added automatically.</td>
</tr>
<tr>
<td>n</td>
<td>Assigns the number of characters read thus far by the call to <code>scanf()</code> to the corresponding argument. The corresponding argument shall be a pointer to an unsigned integer.</td>
</tr>
</tbody>
</table>

An optional assignment-suppressing character (*) can be used after the format specifier to indicate that the conversion specification is to be done, but not saved into a corresponding variable. In this case, no corresponding argument variable should be passed to the `scanf()` function.

A string composed of ordinary non-white space characters is executed by reading the next character of the string. If one of the inputted characters differs from the string,
the function fails and exits. If a white-space character precedes the ordinary non-white space characters, then white-space characters are first read in until a non-white space character is read.

White-space characters are skipped, except for the conversion specifiers [, c or n], unless a white-space character precedes the [ or c specifiers.

<table>
<thead>
<tr>
<th>Availability:</th>
<th>All Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires:</td>
<td>#USE RS232</td>
</tr>
<tr>
<td>Examples:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>char name[2-];</td>
</tr>
<tr>
<td></td>
<td>unsigned int8 number;</td>
</tr>
<tr>
<td></td>
<td>signed int32 time;</td>
</tr>
<tr>
<td></td>
<td>if(scanf(&quot;%u%s%ld&quot;,&amp;number,name,&amp;time))</td>
</tr>
<tr>
<td></td>
<td>printf&quot;\r\nName: %s, Number: %u, Time:</td>
</tr>
<tr>
<td></td>
<td>%ld&quot;,name,number,time);</td>
</tr>
<tr>
<td>Example Files:</td>
<td>None</td>
</tr>
<tr>
<td>Also See:</td>
<td>RS232 I/O Overview, getc(), putc(), printf()</td>
</tr>
</tbody>
</table>
**Built-in Functions**

```c
set_ccp1_compare_time();
set_ccp2_compare_time();
set_ccp3_compare_time();
set_ccp4_compare_time();
set_ccp5_compare_time();
```

**Syntax:**

```
set_ccpx_compare_time(time);
set_ccpx_compare_time(timeA, timeB);
```

**Parameters:**

- **time** - may be a 16 or 32-bit constant or variable. If 16-bit, it sets the CCPxRAL register to the value time and CCPxRBL to zero; used for single edge output compare mode set for 16-bit timer mode. If 32-bit, it sets the CCPxRAL and CCPxRBL register to the value time, CCPxRAL least significant word and CCPRBL most significant word; used for single edge output compare mode set for 32-bit timer mode.

- **timeA** - is a 16-bit constant or variable to set the CCPxRAL register to the value of timeA, used for dual edge output compare and PWM modes.

- **timeB** - is a 16-bit constant or variable to set the CCPxRBL register to the value of timeB, used for dual edge output compare and PWM modes.

**Returns:**

Undefined

**Function:**

This function sets the compare value for the CCP module. If the CCP module is performing a single edge compare in 16-bit mode, then the CCPxRBL register is not used. If 32-bit mode, the CCPxRBL is the most significant word of the compare time. If the CCP module is performing dual edge compare to generate an output pulse, then timeA, CCPxRAL register, signifies the start of the pulse, and timeB, CCPxRBL register signifies the pulse termination time.
**Availability:** Available only on PIC24FxxKMxxx family of devices with a MCCP and/or SCCP modules.

**Requires:** Nothing

**Examples:**

```
setup_ccp1(CCP_COMPARE_PULSE);
set_timer_period_ccp1(800);

set_ccp1_compare_time(200,300);  //generate a pulse
starting at time 300

set_timer_period_ccp1(800);
```

**Example Files:** None

**Also See:** set_pwmX_duty(), setup_ccpX(), set_timer_period_ccpX(), set_timer_ccpX(), get_timer_ccpX(), get_capture_ccpX(), get_captures32_ccpX()

---

**set_cog_blanking()**

**Syntax:**

```
set_cog_blanking(falling_time, rising_time);
```

**Parameters:**

- **falling time** - sets the falling edge blanking time.
- **rising time** - sets the rising edge blanking time.

**Returns:** Nothing

**Function:** To set the falling and rising edge blanking times on the Complementary Output Generator (COG) module. The time is based off the source clock of the COG module, the times are either a 4-bit or 6-bit value, depending on the device, refer to the
device's datasheet for the correct width.

**Availability:** All devices with a COG module.

**Examples:** `set_cog_blanking(10,10);`

**Example Files:** None

**Also See:** `setup_cog()`, `set_cog_phase()`, `set_cog_dead_band()`, `cog_status()`, `cog_restart()`

---

### set_cog_dead_band()

**Syntax:** `set_cog_dead_band(falling_time, rising_time);`

**Parameters:**
- **falling time** - sets the falling edge dead-band time.
- **rising time** - sets the rising edge dead-band time.

**Returns:** Nothing

**Function:** To set the falling and rising edge dead-band times on the Complementary Output Generator (COG) module. The time is based off the source clock of the COG module, the times are either a 4-bit or 6-bit value, depending on the device, refer to the device's datasheet for the correct width.

**Availability:** All devices with a COG module.

**Examples:** `set_cog_dead_band(16,32);`

**Example Files:** None

**Also See:** `setup_cog()`, `set_cog_phase()`, `set_cog_blanking()`, `cog_status()`, `cog_restart()`
**set_cog_phase()**

**Syntax:**
```
set_cog_phase(rising_time);
set_cog_phase(falling_time, rising_time);
```

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>falling time</td>
<td>sets the falling edge phase time.</td>
</tr>
<tr>
<td>rising time</td>
<td>sets the rising edge phase time.</td>
</tr>
</tbody>
</table>

| Returns | Nothing |

| Function | To set the falling and rising edge phase times on the Complementary Output Generator (COG) module. The time is based off the source clock of the COG module, the times are either a 4-bit or 6-bit value, depending on the device. Some devices only have a rising edge delay, refer to the device's datasheet. |

| Availability | All devices with a COG module. |

| Examples | set_cog_phase(10,10); |

| Example Files | None |

| Also See | setup_cog(), set_cog_dead_band(), set_cog_blanking(), cog_status(), cog_restart() |

**set_compare_time()**

**Syntax:**
```
set_compare_time(x, time);
```

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>is 1-8 and defines which output compare module to set time for time is the compare time for the primary compare register.</td>
</tr>
</tbody>
</table>

| Returns | None |

| Function | This function sets the compare value for the ccp module. |

| Availability | Only available on devices with ccp modules. |
set_compare_time( )

Syntax: set_compare_time(x, ocr, [ocrs])

Parameters: x is 1-16 and defines which output compare module to set time for ocr is the compare time for the primary compare register. ocrs is the optional compare time for the secondary register. Used for dual compare mode.

Returns: None

Function: This function sets the compare value for the output compare module. If the output compare module is to perform only a single compare than the ocrs register is not used. If the output compare module is using double compare to generate an output pulse, the ocr signifies the start of the pulse and ocrs defines the pulse termination time.

Availability: Only available on devices with output compare modules.

Requires: Nothing

Examples: // Pin OC1 will be set when timer 2 is equal to 0xF000 setup_timer2(TMR_INTERNAL | TIMER_DIV_BY_8); setup_compare_time(1, 0xF000); setup_compare(1, COMPARE_SET_ON_MATCH | COMPARE_TIMER2);

Example Files: None

Also See: get_capture(), setup_compare(), Output Compare, PWM Overview
**set_dedicated_adc_channel()**

**Syntax:**

```
set_dedicated_adc_channel(core, channel, [differential]);
```

**Parameters:**

- **core** - the dedicated ADC core to setup
- **channel** - the channel assigned to the specified ADC core. Channels are defined in the device's .h file as follows:
  - ADC_CHANNEL_AN0
  - ADC_CHANNEL_AN7
  - ADC_CHANNEL_PGA1
  - ADC_CHANNEL_AN0ALT
  - ADC_CHANNEL_AN1
  - ADC_CHANNEL_AN18
  - ADC_CHANNEL_PGA2
  - ADC_CHANNEL_AN1ALT
  - ADC_CHANNEL_AN2
  - ADC_CHANNEL_AN11
  - ADC_CHANNEL_VREF_BAND_GAP
  - ADC_CHANNEL_AN3
  - ADC_CHANNEL_AN15

  Not all of the above defines can be used with all the dedicated ADC cores. Refer to the device's header for which can be used with each dedicated ADC core.

- **differential** - optional parameter to specify if channel is differential or single-ended. TRUE is differential and FALSE is single-ended.

**Returns:**

Undefined

**Function:**

Sets the channel that will be assigned to the specified dedicated ADC core. Function does not set the channel that will be read with the next call to read_adc(), use set_adc_channel() or read_adc() functions to set the channel that will be read.

**Availability:**

On the dsPIC33EPxxGSxxx family of devices.

**Requires:**

Nothing.
### set_hspwm_override( )

**Syntax:**

```c
set_hspwm_override(unit, setting);
```

**Parameters:**

- **unit** - the High Speed PWM unit to override.
- **settings** - the override settings to use. The valid options vary depending on the device. See the device’s .h file for all options. Some typical options include:
  - `HSPWM_FORCE_H_1`
  - `HSPWM_FORCE_H_0`
  - `HSPWM_FORCE_L_1`
  - `HSPWM_FORCE_L_0`

**Returns:**

Undefined

**Function:**

Setup and High Speed PWM override settings.

**Availability:**

Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**

None

**Examples:**

```c
setup_hspwm_override(1, HSPWM_FORCE_H_1 | HSPWM_FORCE_L_0);
```

**Example Files:**

None

**Also See:**

- setup_hspwm_unit()
- set_hspwm_phase()
- set_hspwm_duty()
- set_hspwm_event()
- setup_hspwm_blanking()
- setup_hspwm_trigger()
- get_hspwm_capture()
- setup_hspwm_chop_clock()
- setup_hspwm_unit_chop_clock()
- setup_hspwm()
- setup_hspwm_secondary()
set_hspwm_phase( )

Syntax:  

```c
set_hspwm_phase(unit, primary, [secondary]);
```

Parameters:  

- **unit** - The High Speed PWM unit to set.
- **primary** - A 16-bit constant or variable to set the primary duty cycle.
- **secondary** - An optional 16-bit constant or variable to set the secondary duty cycle. Secondary duty cycle is only used in Independent PWM mode. Not available on all devices, refer to device datasheet for availability.

Returns:  

undefined

Function:  

Sets up the specified High Speed PWM unit.

Availability:  

Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

Requires:  

Constants are defined in the device’s .h file

Examples:  

```c
set_hspwm(1,0x1000,0x8000);
```

Example Files:  

None

Also See:  

setup_hspwm_unit(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_blanking(), setup_hspwm_trigger(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock(), setup_hspwm(), setup_hspwm_secondary()

---

set_input_level_x()

Syntax:  

```c
set_input_level_a(value)
set_input_level_b(value)
set_input_level_v(value)
```
**Built-in Functions**

```plaintext
set_input_level_d(value)
set_input_level_e(value)
set_input_level_f(value)
set_input_level_g(value)
set_input_level_h(value)
set_input_level_j(value)
set_input_level_k(value)
set_input_level_l(value)
```

**Parameters:** value is an 8-bit int with each bit representing a bit of the I/O port.

**Returns:** undefined

**Function:** These functions allow the I/O port Input Level Control (INLVLx) registers to be set. Each bit in the value represents one pin. A 1 sets the corresponding pin's input level to Schmitt Trigger (ST) level, and a 0 sets the corresponding pin's input level to TTL level.

**Availability:** All devices with ODC registers, however not all devices have all I/O ports and not all devices port's have a corresponding ODC register.

**Requires:** Nothing

**Examples:**
- `set_input_level_a(0x0);` //sets PIN_A0 input level to ST and all other
- `//PORTA pins to TTL level

**Example Files:** None

**Also See:** output_high(), output_low(), output_bit(), output_x(), General Purpose I/O

---

**set_motor_pwm_duty( )**

**Syntax:**

```
set_motor_pwm_duty(pwm, group, time);
```

**Parameters:**

- **pwm** - Defines the pwm module used.
- **group** - Output pair number 1, 2 or 3.
- **time** - The value set in the duty cycle register.

**Returns:** void

**Function:** Configures the motor control PWM unit duty.
### set_motor_pwm_event()

**Syntax:**
```c
set_motor_pwm_event(pwm, time);
```

**Parameters:**
- `pwm` - Defines the pwm module used.
- `time` - The value in the special event comparator register used for scheduling other events.

**Returns:**
`void`

**Function:**
Configures the PWM event on the motor control unit.

**Availability:**
Devices that have the motor control PWM unit.

**Requires:**
None

**Examples:**
```c
set_motor_pwm_event(pwm, time);
```

**Example Files:**
None

**Also See:**
- `get_motor_pwm_count()`, `set_motor_pwm_event()`, `set_motor_unit()`, `setup_motor_pwm()`,
- `set_motor_pwm_duty()`
### set_motor_unit( )

**Syntax:**

```c
set_motor_unit(pwm, unit, options, active_deadtime, inactive_deadtime);
```

**Parameters:**

- `pwm`: Defines the pwm module used
- `Unit`: This will select Unit A or Unit B
- `options`: The mode of the power PWM module. See the devices .h file for all options
- `active_deadtime`: Set the active deadtime for the unit
- `inactive_deadtime`: Set the inactive deadtime for the unit

**Returns:**

`void`

**Function:**

Configures the motor control PWM unit.

**Availability:**

Devices that have the motor control PWM unit

**Requires:**

None

**Examples:**

```c
set_motor_unit(pwm, unit, MPWM_INDEPENDENT | MPWM_FORCE_L_1, active_deadtime, inactive_deadtime);
```

**Example Files:**

None

**Also See:**

- `get_motor_pwm_count()
- `set_motor_pwm_event()
- `set_motor_pwm_duty()
- `setup_motor_pwm()`

### set_nco_inc_value( )

**Syntax:**

```c
set_nco_inc_value(value);
```

**Parameters:**

- `value`: value to set the NCO increment registers

**Returns:**

Undefined

**Function:**

Sets the value that the NCO's accumulator will be incremented by on each clock pulse. The increment registers are double buffered so the
new value won't be applied until the accumulator rolls-over.

Availability: On devices with a NCO module.

Examples: set_nco_inc_value(inc_value);  //sets the new increment value

Example Files: None

Also See: setup_nco(), get_nco_accumulator(), get_nco_inc_value()

set_open_drain_x(value)

Syntax:

set_open_drain_a(value)
set_open_drain_b(value)
set_open_drain_c(value)
set_open_drain_d(value)
set_open_drain_e(value)
set_open_drain_f(value)
set_open_drain_g(value)
set_open_drain_h(value)
set_open_drain_j(value)
set_open_drain_k(value)

Parameters: value – is an 8-bit int with each bit representing a bit of the I/O port.
value – is a bitmap corresponding to the pins of the port. Setting a bit causes the corresponding pin to act as an open-drain output.

Returns: Nothing

Function: These functions allow the I/O port Open-Drain Control (ODCONx) registers to be set. Each bit in the value represents one pin. A 1 sets the corresponding pin to act as an open-drain output, and a 0 sets the corresponding pin to act as a digital output.

Enables/Disables open-drain output capability on port pins. Not all ports or port pins have open-drain capability, refer to devices data sheet for port and pin availability.

Availability: Nothing.

Examples: set_open_drain_a(0x01);  //makes PIN_A0 an open-drain output.
set_open_drain_b(0x001);  //enables open-drain output on
set_pulldown( )

Syntax: set_Pulldown(state [, pin])

Parameters: Pins are defined in the devices .h file. If no pin is provided in the function call, then all of the pins are set to the passed in state.  

State is either true or false.

Returns: undefined

Function: Sets the pin's pull down state to the passed in state value. If no pin is included in the function call, then all valid pins are set to the passed in state.

Availability: All devices that have pull-down hardware.

Requires: Pin constants are defined in the devices .h file.

Examples:

```c
set_pulldown(true, PIN_B0);
    //Sets pin B0’s pull down state to true

set_pulldown(false);
    //Sets all pin's pull down state to false
```

Example Files: None

Also See: None
# set_pullup()

**Syntax:**

```
set_Pullup(state, [pin])
```

**Parameters:**

- Pins are defined in the devices .h file. If no pin is provided in the function call, then all of the pins are set to the passed in state.

  - **State** is either true or false.

  Pins are defined in the devices .h file. The actual number is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #DEFINE PIN_A3 43. The pin could also be a variable that has a value equal to one of the predefined pin constants. Note if no pin is provided in the function call, then all of the pins are set to the passed in state.

  - **State** is either true or false.

**Returns:**

undefined

**Function:**

Sets the pin's pull up state to the passed in state value. If no pin is included in the function call, then all valid pins are set to the passed in state.

**Availability:**

All devices.

**Requires:**

Pin constants are defined in the devices .h file.

**Examples:**

```c
set_pullup(true, PIN_B0);
  //Sets pin B0's pull up state to true

set_pullup(false);
  //Sets all pin's pull up state to false
```

**Example Files:**

None

**Also See:**

None
### set_pwmX_duty( )

**Syntax:**

`set_pwmX_duty(value)`

**Parameters:**

`value` may be an 8 or 16 bit constant or variable.

**Returns:**

undefined

**Function:**

PIC24FxxKLxxx devices, writes the 10-bit value to the PWM to set the duty. An 8-bit value may be used if the most significant bits are not required. The 10-bit value is then used to determine the duty cycle of the PWM signal as follows:

- Duty cycle = `value / [4 * (PRx + 1)]`
  
  Where PRx is the maximum value timer 2 or 4 will count to before rolling over.

PIC24FxxKMxxx devices, wires the 16-bit value to the PWM to set the duty. The 16-bit value is then used to determine the duty cycle of the PWM signal as follows:

- Duty cycle = `value/(CCPxPRL+1)`
  
  Where CCPxPRL is the maximum value timer 2 will count to before toggling the output pin.

**Availability:**

This function is only available on devices with MCCP and/or SCCP modules.

**Requires:**

None

**Examples:**

**PIC24FxxKLxxx Devices:**

```c
// 32 MHz clock
unsigned int16 duty;

setup_timer2(T2_DIV_BY_4, 199, 1);  //period=50us
setup_ccp1(CCP_PWM);

duty=400;                           //duty=400/[4*(199+1)]=0.5=50%
```

set_pwm1_duty(duty);

PIC24FxxKMxxx Devices:
// 32 MHz clock
unsigned int16 duty;

setup_ccp1(CCP_PWM);
set_timer_period_ccp1(799);  //period=50us

duty=400;
    //duty=400/(799+1)=0.5=50%
set_pwm1_duty(duty);

Example Files:  ex_pwm.c
Also See: setup_ccpX(), set_ccpX_compare_time(), set_timer_period_ccpX(),
set_timer_ccpX(), get_timer_ccpX(), get_capture_ccpX(),
getcaptures32_ccpX()

set_pwm1_offset()  set_pwm2_offset()
set_pwm3_offset()  set_pwm4_offset()
set_pwm5_offset()  set_pwm6_offset()

Syntax:  set_pwm1_offset (value)
         set_pwm2_offset (value)
         set_pwm3_offset (value)
         set_pwm4_offset (value)
         set_pwm5_offset (value)
         set_pwm6_offset (value)

Parameters:  value - 16-bit constant or variable.
Returns:  undefined.
Function:  Writes the 16-bit to the PWM to set the offset. The offset is used to adjust
the waveform of a slave PWM module relative to the waveform of a master
PWM module.
Availability: Devices with a 16-bit PWM module.
Built-in Functions

**Requires:** Nothing

**Examples:**
```
set_pwm1_offset(0x0100);
set_pwm1_offset(offset);
```

**Example Files:**
```
setup_pwm(), set_pwm_duty(), set_pwm_period(), clear_pwm_interrupt(),
set_pwm_phase(), enable_pwm_interrupt(), disable_pwm_interrupt(),
pwm_interrupt_active()
```

**Syntax:**
```
set_pwm1_period(value)
set_pwm2_period(value)
set_pwm3_period(value)
set_pwm4_period(value)
set_pwm5_period(value)
set_pwm6_period(value)
```

**Parameters:**
- `value` - 16-bit constant or variable.

**Returns:** undefined.

**Function:** Writes the 16-bit to the PWM to set the period. When the PWM module is set-up for standard mode it sets the period of the PWM signal. When set-up for set on match mode, it sets the maximum value at which the phase match can occur. When in toggle on match and center aligned modes it sets the maximum value the PWMxTMR will count to, the actual period of PWM signal will be twice what the period was set to.

**Availability:** Devices with a 16-bit PWM module.

**Requires:** Nothing

**Examples:**
```
set_pwm1_period(0x8000);
set_pwm1_period(period);
```

**Example**
set_pwm1_phase()  set_pwm2_phase()
set_pwm3_phase()  set_pwm4_phase()
set_pwm5_phase()  set_pwm6_phase()

Syntax:

set_pwm1_phase(value)
set_pwm2_phase(value)
set_pwm3_phase(value)
set_pwm4_phase(value)
set_pwm5_phase(value)
set_pwm6_phase(value)

Parameters:

value - 16-bit constant or variable.

Returns:

undefined.

Function:

Writes the 16-bit to the PWM to set the phase. When the PWM module is set-up for standard mode the phase specifies the start of the duty cycle, when in set on match mode it specifies when the output goes high, and when in toggle on match mode it specifies when the output toggles. Phase is not used when in center aligned mode.

Availability:

Devices with a 16-bit PWM module.

Requires:

Nothing

Examples:

set_pwm1_phase(0);
set_pwm1_phase(phase);
## set_open_drain_x()

**Syntax:**
- `set_open_drain_a(value)`
- `set_open_drain_b(value)`
- `set_open_drain_v(value)`
- `set_open_drain_d(value)`
- `set_open_drain_e(value)`
- `set_open_drain_f(value)`
- `set_open_drain_g(value)`
- `set_open_drain_h(value)`
- `set_open_drain_i(value)`
- `set_open_drain_j(value)`
- `set_open_drain_k(value)`

**Parameters:**
- `value` is an 16-bit int with each bit representing a bit of the I/O port.

**Returns:**
- undefined

**Function:**
These functions allow the I/O port Open-Drain Control (ODC) registers to be set. Each bit in the value represents one pin. A 1 sets the corresponding pin to act as an open-drain output, and a 0 sets the corresponding pin to act as a digital output.

**Availability:**
All devices with ODC registers, however not all devices have all I/O ports and not all devices port's have a corresponding ODC register.

**Requires:**
Nothing

**Examples:**
- `set_open_drain_a(0x0001);`  // makes PIN_A0 an open-drain output

**Example Files:**
None

**Also See:**
`output_high()`, `output_low()`, `output_bit()`, `output_x()`, General Purpose I/O
**set_rtcc()**  **set_timer0()**  **set_timer1()**  **set_timer2()**  **set_timer3()**  **set_timer4()**  **set_timer5()**

| Syntax: | set_timer0(value)  or  set_rtcc(value)  
|         | set_timer1(value)  
|         | set_timer2(value)  
|         | set_timer3(value)  
|         | set_timer4(value)  
|         | set_timer5(value)  |

| Parameters: | Timers 1 & 5 get a 16 bit int.  
|             | Timer 2 and 4 gets an 8 bit int.  
|             | Timer 0 (AKA RTCC) gets an 8 bit int except on the PIC18XXX where it needs a 16 bit int.  
|             | Timer 3 is 8 bit on PIC16 and 16 bit on PIC18 |

| Returns: | undefined |

| Function: | Sets the count value of a real time clock/counter. RTCC and Timer0 are the same. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2...) |

| Availability: | Timer 0 - All devices  
|               | Timers 1 & 2 - Most but not all PCM devices  
|               | Timer 3 - Only PIC18XXX and some pick devices  
|               | Timer 4 - Some PCH devices  
|               | Timer 5 - Only PIC18XX31 |

| Requires: | Nothing |

| Examples: | // 20 mhz clock, no prescaler, set timer 0  
|           | // to overflow in 35us  
|           | set_timer0(81);  // 256-(.000035/(4/20000000)) |

**Example Files:**  **ex_patg.c**  

**Also See:**  **set_timer1()**,  **get_timerX()**  Timer0 Overview, Timer1Overview, Timer2 Overview, Timer5 Overview
set_ticks()  

**Syntax:**

```
set_ticks([stream],value);
```

**Parameters:**

- `stream` – optional parameter specifying the stream defined in #USE TIMER
- `value` – a 8, 16, 32 or 64 bit integer, specifying the new value of the tick timer. (int8, int16, int32 or int64)

**Returns:**

`void`

**Function:**

Sets the new value of the tick timer. Size passed depends on the size of the tick timer.

**Availability:**

All devices.

**Requires:**

#USE TIMER(options)

**Examples:**

```
#USE TIMER(TIMER=1,TICK=1ms,BITS=16,NOISR)

void main(void) {
    unsigned int16 value = 0x1000;
    set_ticks(value);
}
```

**Example Files:**

None

**Also See:**

#USE TIMER, get_ticks()

setup_sd_adc_calibration()  

**Syntax:**

```
setup_sd_adc_calibration(model);
```

**Parameters:**

- `mode` - selects whether to enable or disable calibration mode for the SD ADC module. The following defines are made in the device's .h file:
  1. `SDADC_START_CALIBRATION_MODE`
  2. `SDADC_END_CALIBRATION_MODE`

**Returns:**

Nothing
## Function:
To enable or disable calibration mode on the Sigma-Delta Analog to Digital Converter (SD ADC) module. This can be used to determine the offset error of the module, which then can be subtracted from future readings.

### Availability:
Only devices with a SD ADC module.

### Examples:
signed int 32 result, calibration;

set_sd_adc_calibration(SDADC_START_CALIBRATION_MODE);
calibration = read_sd_adc();
set_sd_adc_calibration(SDADC_END_CALIBRATION_MODE);

result = read_sd_adc() - calibration;

## set_sd_adc_channel()

### Syntax:
```c
setup_sd_adc(channel);
```

### Parameters:
- **channel**: sets the SD ADC channel to read. Channel can be 0 to read the difference between CH0+ and CH0-, 1 to read the difference between CH1+ and CH1-, or one of the following:
  1. SDADC_CH1SE_SVSS
  2. SDADC_REFERENCE

### Returns:
Nothing

### Function:
To select the channel that the Sigma-Delta Analog to Digital Converter (SD ADC) performs the conversion on.

### Availability:
Only devices with a SD ADC module.

### Examples:
set_sd_adc_channel(0);

### Example Files:
None

### Also See:
- setup_sd_adc()
- read_sd_adc()
- set_sd_adc_calibration()
### set_timerA()

**Syntax:**

```
set_timerA(value);
```

**Parameters:**

An 8 bit integer. Specifying the new value of the timer. (int8)

**Returns:**

undefined

**Function:**

Sets the current value of the timer. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2, …).

**Availability:**

This function is only available on devices with Timer A hardware.

**Requires:**

Nothing

**Examples:**

```c
// 20 mhz clock, no prescaler, set timer A
// to overflow in 35us

set_timerA(81); // 256-(000035/(4/2000000))
```

**Example Files:**

none

**Also See:**

get_timerA(), setup_timer_A(), TimerA Overview

### set_timerB()

**Syntax:**

```
set_timerB(value);
```

**Parameters:**

An 8 bit integer. Specifying the new value of the timer. (int8)

**Returns:**

undefined

**Function:**

Sets the current value of the timer. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2, …).

**Availability:**

This function is only available on devices with Timer B hardware.

**Requires:**

Nothing
Examples: // 20 mhz clock, no prescaler, set timer B
// to overflow in 35us

        set_timerB(81); // 256-(.000035/(4/20000000))

Example
Files:    none
Also See: get_timerB(), setup_timer_B(), TimerB Overview

set_timerx( )

Syntax:    set_timerX(value)

Parameters: A 16 bit integer, specifying the new value of the timer. (int16)
Returns:   void
Function:  Allows the user to set the value of the timer.
Availability: This function is available on all devices that have a valid timerX.
Requires:  Nothing
Examples:  if(EventOccured())
            set_timer2(0);//reset the timer.

Example
Files:    None
Also See: Timer Overview, setup_timerX(), get_timerXY(), set_timerX(),
          set_timerXY()

set_timerxy( )

Syntax:    set_timerXY(value)

Parameters: A 32 bit integer, specifying the new value of the timer. (int32)
Returns:   void
Function:  Retrieves the 32 bit value of the timers X and Y, specified by XY(which
          may be 23, 45, 67 and 89)
Availability: This function is available on all devices that have a valid 32 bit enabled
Built-in Functions

Timers 2 & 3, 4 & 5, 6 & 7 and 8 & 9 may be used. The target device must have one of these timer sets. The target timers must be enabled as 32 bit.

Requires: Nothing

Examples:

```c
if(get_timer45() == THRESHOLD)
    set_timer(THRESHOLD + 0x1000);//skip those timer values
```

Example Files: None

Also See: Timer Overview, setup_timerX(), get_timerXY(), set_timerX(), set_timerXY()

<table>
<thead>
<tr>
<th>Syntax</th>
<th>set_timer0( )</th>
<th>set_timer1( )</th>
<th>set_timer2( )</th>
<th>set_timer3( )</th>
<th>set_timer4( )</th>
<th>set_timer5( )</th>
</tr>
</thead>
</table>
| Parameters   | Timers 1 & 5 get a 16 bit int.  
Timer 2 and 4 gets an 8 bit int.  
Timer 0 (AKA RTCC) gets an 8 bit int except on the PIC18XXX where it needs a 16 bit int.  
Timer 3 is 8 bit on PIC16 and 16 bit on PIC18 |
| Returns      | undefined     |
| Function     | Sets the count value of a real time clock/counter. RTCC and Timer0 are the same. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2...) |
| Availability | Timer 0 - All devices  
Timers 1 & 2 - Most but not all PCM devices  
Timer 3 - Only PIC18XXX and some pick devices |
Timer 4 - Some PCH devices
Timer 5 - Only PIC18XX31

Requires: Nothing

Examples:

```c
// 20 mhz clock, no prescaler, set timer 0
// to overflow in 35us
set_timer0(81);       // 256-(.000035/(4/20000000))
```

Example Files: `ex_patg.c`

Also See: `set_timer1()`, `get_timerX()` Timer0 Overview, Timer1Overview, Timer2 Overview, Timer5 Overview

### set_timer_ccp1() set_timer_ccp2()
### set_timer_ccp3() set_timer_ccp4()
### set_timer_ccp5()

**Syntax:**

```c
set_timer_ccpx(time);
set_timer_ccpx(timeL, timeH);
```

**Parameters:**

- `time` - may be a 32-bit constant or variable. Sets the timer value for the CCPx module when in 32-bit mode.
- `timeL` - may be a 16-bit constant or variable to set the value of the lower timer when CCP module is set for 16-bit mode.
- `timeH` - may be a 16-bit constant or variable to set the value of the upper timer when CCP module is set for 16-bit mode.

**Returns:** Undefined

**Function:** This function sets the timer values for the CCP module. `TimeH` is optional parameter when using 16-bit mode, defaults to zero if not specified.

**Availability:** Available only on PIC24FxxKMxxx family of devices with a MCCP and/or SCCP modules.
### Built-in Functions

**Requires:** Nothing  
**Examples:**
- `setup_ccp1(CCP_TIMER);` //set for dual timer mode  
- `set_timer_ccp1(100,200);` //set lower timer value to 100 and upper timer //value to 200  
**Example Files:** None  
**Also See:**
- `set_pwmX_duty()`, `setup_ccpX()`, `set_cpx_compare_time()`, `get_capture_cpx()`, `set_timer_period_cpx()`, `get_timer_cpx()`, `get_captures32_cpx()`

---

**set_timer_period_cpx1()**
**set_timer_period_cpx2()**
**set_timer_period_cpx3()**
**set_timer_period_cpx4()**
**set_timer_period_cpx5()**

**Syntax:**
- `set_timer_period_cpxx(time);`  
- `set_timer_period_cpxx(timeL, timeH);`

**Parameters:**
- `time` - may be a 32-bit constant or variable. Sets the timer period for the CCPx module when in 32-bit mode.  
- `timeL` - is a 16-bit constant or variable to set the period of the lower timer when CCP module is set for 16-bit mode.  
- `timeH` - is a 16-bit constant or variable to set the period of the upper timer when CCP module is set for 16-bit mode.

**Returns:** Undefined
Function: This function sets the timer periods for the CCP module. When setting up CCP module in 32-bit function is only needed when using Timer mode. Period register are not used when module is setup for 32-bit compare mode, period is always 0xFFFFFFFF. TimeH is optional parameter when using 16-bit mode, default to zero if not specified.

Availability: Available only on PIC24FxxKMxxx family of devices with a MCCP and/or SCCP modules.

Requires: Nothing

Examples: setup_ccp1(CCP_TIMER); //set for dual timer mode
set_timer_period_ccp1(800,2000); //set lower timer period to 800 and
2000 //upper timer period to

Example Files: None

Also See: set_pwmX_duty(), setup_ccpX(), set_ccpX_compare_time(), set_timer_ccpX(), get_timer_ccpX(), get_capture_ccpX(), get_captures32_ccpX()

---

**set_tris_x( )**

**Syntax:**

```
set_tris_a (value)
set_tris_b (value)
set_tris_c (value)
set_tris_d (value)
set_tris_e (value)
set_tris_f (value)
set_tris_g (value)
set_tris_h (value)
set_tris_j (value)
set_tris_k (value)
```

**Parameters:**

*value* is an 16 bit int with each bit representing a bit of the I/O port.

**Returns:** undefined
### Built-in Functions

| Function: | These functions allow the I/O port direction (TRI-State) registers to be set. This must be used with FAST.IO and when I/O ports are accessed as memory such as when a # word directive is used to access an I/O port. Using the default standard I/O the built in functions set the I/O direction automatically. Each bit in the value represents one pin. A 1 indicates the pin is input and a 0 indicates it is output. |
| Availability: | All devices (however not all devices have all I/O ports) |
| Requires: | Nothing |
| Examples: | \[
\text{SET TRIS B( 0x0F );}
\]
// B7,B6,B5,B4 are outputs
// B15,B14,B13,B12,B11,B10,B9,B8, B3,B2,B1,B0 are inputs |
| Example Files: | lcd.c |
| Also See: | #USE FAST IO, #USE FIXED IO, #USE STANDARD IO, General Purpose I/O |

### set_uart_speed( )

| Syntax: | \text{set_uart_speed } (\text{baud}, [\text{stream, clock}]) |
| Parameters: | \text{baud} is a constant representing the number of bits per second.\text{stream} is an optional stream identifier.\text{clock} is an optional parameter to indicate what the current clock is if it is different from the #use delay value |
| Returns: | undefined |
| Function: | Changes the baud rate of the built-in hardware RS232 serial port at runtime. |
| Availability: | This function is only available on devices with a built in UART. |
| Requires: | #USE RS232 |
Examples:  // Set baud rate based on setting // of pins B0 and B1

switch( input_b() & 3 ) {
    case 0 : set_uart_speed(2400); break;
    case 1 : set_uart_speed(4800); break;
    case 2 : set_uart_speed(9600); break;
    case 3 : set_uart_speed(19200); break;
}

Examples:  // USE RS232, putc(), getc(), setup uart(), RS232 I/O Overview

Example Files:  loader.c

Also See:  #USE RS232, putc(), getc(), setup uart(), RS232 I/O Overview

---

**setjmp( )**

**Syntax:**  
result = setjmp (env)

**Parameters:**  
env: The data object that will receive the current environment

**Returns:**  
If the return is from a direct invocation, this function returns 0.
If the return is from a call to the longjmp function, the setjmp function returns a nonzero value and it's the same value passed to the longjmp function.

**Function:**  
Stores information on the current calling context in a data object of type jmp_buf and which marks where you want control to pass on a corresponding longjmp call.

**Availability:**  
All devices

**Requires:**  
#include <setjmp.h>

**Examples:**  
result = setjmp(jmpbuf);

**Example Files:**  None

**Also See:**  longjmp()
# Built-in Functions

## setup_adc(mode)  setup_adc2(mode)

**Syntax:**

```c
setup_adc (mode);
setup_adc2 (mode);
```

**Parameters:**

*mode* - Analog to digital mode. The valid options vary depending on the device. See the devices .h file for all options. Some typical options include:

- ADC_OFF
- ADC_CLOCK_INTERNAL
- ADC_CLOCK_DIV_32
- ADC_CLOCK_INTERNAL – The ADC will use an internal clock
- ADC_CLOCK_DIV_32 – The ADC will use the external clock scaled down by 32
- ADC_TAD_MUL_16 – The ADC sample time will be 16 times the ADC conversion time

**Returns:**

undefined

**Function:**

Configures the ADC clock speed and the ADC sample time. The ADC converters have a maximum speed of operation, so ADC clock needs to be scaled accordingly. In addition, the sample time can be set by using a bitwise OR to concatenate the constant to the argument.

**Availability:**

Only the devices with built-in analog to digital converter.

**Requires:**

Constants are defined in the devices .h file.

**Examples:**

```c
setup_adc_ports( ALL_ANALOG );
setup_adc(ADC_CLOCK_INTERNAL);
set_adc_channel( 0 );
value = read_adc();
setup_adc( ADC_OFF );
```

**Example Files:**

`ex_admm.c`

**Also See:**

`setup_adc_ports()`, `set_adc_channel()`, `read_adc()`, `#DEVICE`, `ADC Overview`,

see header file for device selected
### setup_adc_ports( )

#### Syntax:

```plaintext
setup_adc_ports (ports, reference)
```

#### Parameters:

- **value**: a constant defined in the device's .h file
- **ports**: is a constant specifying the ADC pins to use
- **reference**: is an optional constant specifying the ADC reference to use. By default, the reference voltage are Vss and Vdd

#### Returns:

undefined

#### Function:

Sets up the ADC pins to be analog, digital, or a combination and the voltage reference to use when computing the ADC value. The allowed analog pin combinations vary depending on the chip and are defined by using the bitwise OR to concatenate selected pins together. Check the device include file for a complete list of available pins and reference voltage settings. The constants ALL_ANALOG and NO_ANALOGS are valid for all chips.

Some other example pin definitions are:

- `sAN1 | sAN2`: AN1 and AN2 are analog, remaining pins are digital
- `sAN0 | sAN3`: AN0 and AN3 are analog, remaining pins are digital

#### Availability:

This function is only available on devices with A/D hardware. This function is only available on devices with built-in A/D converters.

#### Requires:

Constants are defined in the device's .h file.

#### Examples:

// All pins analog (that can be)

```plaintext
setup_adc_ports(ALL_ANALOG);
```

// Pins A0, A1, and A3 are analog and all others are digital.
// The +5V is used as a reference.

```plaintext
setup_adc_ports(RA0_RA1_RA3_ANALOG);
```

// Pins A0 and A1 are analog. Pin RA3 is use for the reference voltage
// and all other pins are digital.
setup_adc_ports(A0_RA1_ANALOGRA3_REF);

// Set all ADC pins to analog mode.

setup_adc_ports(ALL_ANALOG);

// Pins AN0, AN1, and AN3 are analog and all other pins are
digital.

setup_adc_ports(sAN0|sAN1|sAN3);

// Pins AN0 and AN1 are analog. The VrefL pin and Vdd are
// voltage references.

setup_adc_ports(sAN0|sAN1, VREF_VDD);

Example Files:  ex_admm.c

Also See:  #USE RS232, putc(), getc(), setup uart(), RS232 I/O Overview,

setup_adc_reference( )

Syntax:  setup_adc_reference(reference)

Parameters:  reference - the voltage reference to set the ADC. The valid options
depend on the device, see the device's .h file for all options. Typical
options include:
  · VSS_VDD
  · VSS_VREF
  · VREF_VREF
  · VREF_VDD

Returns:  undefined

Function:  To set the positive and negative voltage reference for the Analog to
Digital Converter (ADC) uses.

Availability:  Only on devices with an ADC and has ANSELx, x being the port letter,
registers for setting which pins are analog or digital.

Requires:  Nothing

Examples:  set_adc_reference(VSS_VREF);
Example Files: set_analog_pins(), set_adc_channel(), read_adc(), setup_adc(), setup_adc_ports(), ADC Overview

setup_at( )

Syntax: setup_at(settings);

Parameters: settings - the setup of the AT module. See the device's header file for all options. Some typical options include:
- AT_ENABLED
- AT_DISABLED
- AT_MULTI_PULSE_MODE
- AT_SINGLE_PULSE_MODE

Returns: Nothing

Function: To setup the Angular Timer (AT) module.

Availability: All devices with an AT module.

Requires: Constants defined in the device's .h file

Examples: setup_at(AT_ENABLED|AT_MULTI_PULSE_MODE|AT_INPUT_ATIN);

Example Files: None

Also See: at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status()
### setup_capture( )

**Syntax:**

```c
setup_capture(x, mode)
```

**Parameters:****

- `x` is 1-16 and defines which input capture module is being configured
- `mode` is defined by the constants in the devices .h file

**Returns:**

None

**Function:**

This function specifies how the input capture module is going to function based on the value of mode. The device specific options are listed in the device .h file.

**Availability:**

Only available on devices with Input Capture modules

**Requires:**

None

**Examples:**

```c
setup_timer3(TMR_INTERNAL | TMR_DIV_BY_8);
setup_capture(2, CAPTURE_FE | CAPTURE_TIMER3);
while(TRUE) {
    timerValue = get_capture(2, TRUE);
    printf("Capture 2 occurred at: %LU", timerValue);
}
```

**Example Files:**

None

**Also See:**

`get_capture()`, `setup_compare()`, `Input Capture Overview`

---

### setup_ccp1()  setup_ccp2()
### setup_ccp3()  setup_ccp4()
### setup_ccp5()  setup_ccp6()

**Syntax:**

```c
setup_ccpx(mode,[pwm]);PIC24FxxKLxxx devices
setup_ccpx(mode1,[mode2],[mode3],[dead_time]);PIC24FxxKMxxx devices
```

**Parameters:**

- `mode` and `mode1` are constants used for setting up the CCP module.
- Valid constants are defined in the device’s .h file, refer to the device’s .h
file for all options. Some typical options are as follows:
- CCP_OFF
- CCP_COMPARE_INT_AND_TOGGLE
- CCP_CAPTURE_FE
- CCP_CAPTURE_RE
- CCP_CAPTURE_DIV_4
- CCP_CAPTURE_DIV_16
- CCP_COMPARE_SET_ON_MATCH
- CCP_COMPARE_CLR_ON_MATCH
- CCP_COMPARE_INT
- CCP_COMPARE_RESET_TIMER
- CCP_PWM

**mode2** is an optional parameter for setting up more settings of the CCP module. Valid constants are defined in the device's .h file, refer to the device's .h file for all options.

**mode3** is an optional parameter for setting up more settings of the CCP module. Valid constants are defined in the device's .h file, refer to the device's .h file for all options.

**pwm** is an optional parameter for devices that have an ECCP module. This parameter allows setting the shutdown time. The value may be 0-255.

**dead_time** is an optional parameter for setting the dead time when the CCP module is operating in PWM mode with complementary outputs. The value may be 0-63, 0 is the default setting if not specified.

**Returns:** Undefined

**Function:** Initializes the CCP module. For PIC24FxxKLxxx devices the CCP module can operate in three modes (Capture, Compare or PWM).

- **Capture Mode** - the value of Timer 3 is copied to the CCPRxH and CCPRxL registers when an input event occurs.
- **Compare Mode** - will trigger an action when Timer 3 and the CCPRxL and CCPRxH registers are equal.
- **PWM Mode** - will generate a square wave, the duty cycle of the signal can be adjusted using the CCPRxL register and the DCxB bits of the CCPxCON register. The function `set_pwmx_duty()` is provided for setting the duty cycle when in PWM mode.
Built-in Functions

mode.

PIC24FxxKMxxx devices, the CCP module can operate in four mode (Timer, Capture, Compare or PWM). IN Timer mode, it functions as a timer. The module has to basic modes, it can functions as two independent 16-bit timers/counters or as a single 32-bit timer/counter. The mode it operates in is controlled by the option CCP_TIMER_32_BIT, with the previous options added, the module operates as a single 32-bit timer, and if not added, it operates as two 16-bit timers. The function set_timer_period_ccpx() is provided to set the period(s) of the timer, and the functions set_timer_ccpx() and get_timer_ccpx() are provided to set and get the current value of the timer(s).

In Capture mode, the value of the timer is captured when an input event occurs, it can operate in either 16-bit or 32-bit mode. The functions get_capture_ccpx() and get_capture32_ccpx() are provided to get the last capture value.

In Compare and PWM modes, the value of the timers is compared to one or two compare registers, depending on its mode of operation, to generate a single output transition or a train of output pulses. For signal output edge modes, CCPCOMPARE_SET_ON_MATCH, CCP_COMPARE_CLR_ON_MATCH, and CCP COMPARE_TOGGLE, the module can operate in 16 or 32-bit mode, all other modes can only operate in 16-bit mode. However, when in 32-bit mode the timer source will only rollover when it reaches 0xFFFFFFFF or when reset from an external synchronization source. Therefore, a period of less than 0xFFFFFFFF is needed, as it requires an external synchronization source to reset the timer. The functions set_ccpx_compare_time() and set_pwmx_duty() are provided for setting the compare registers.

**Availability:** Only on devices with the MCCP and/or SCCP modules.

**Requires:** Constants are defined in the devices .h file.

**Examples:**

```c
setup_ccp1(CCP_CAPTURE_FE);
setup_ccp1(CCP_COMPARE.Toggle);
setup_ccp1(CCP_PWM);
```

**Example Files:**

`ex_pwm.c`, `ex_ccpmp.c`, `ex_ccp1s.c`

**Also See:**

`set_pwmX_duty()`, `set_ccpX_compare_time()`, `set_timer_period_ccpX()`,
`set_timer_ccpX()`, `get_timer_ccpX()`, `get_capture_ccpX()`. 
```markdown
## get_captures32_cpx()

### setup_clc1()

**Syntax:**

```c
setup_clc1(mode);
setup_clc2(mode);
setup_clc3(mode);
setup_clc4(mode);
```

**Parameters:**
- **mode** – The mode to setup the Configurable Logic Cell (CLC) module into. See the device's .h file for all options. Some typical options include:
  - CLC_ENABLED
  - CLC_OUTPUT
  - CLC_MODE_AND_OR
  - CLC_MODE_OR_XOR

**Returns:**
- Undefined.

**Function:**
Sets up the CLC module to performed the specified logic. Please refer to the device datasheet to determine what each input to the CLC module does for the select logic function.

**Availability:**
- On devices with a CLC module.

**Returns:**
- Undefined.

**Examples:**

```c
setup_clc1(CLC_ENABLED | CLC_MODE_AND_OR);
```

**Example Files:**
- None

**Also See:**
- clcx_setup_gate(), clcx_setup_input()
### `setup_comparator( )`

**Syntax:**

```
setup_comparator (mode)
```

**Parameters:**

*`mode`* is a bit-field comprised of the following constants:

- `NC_NC_NC_NC`
- `A4_A5_NC_NC`
- `A4_VR_NC_NC`
- `A5_VR_NC_NC`
- `NC_NC_A2_A3`
- `NC_NC_A2_VR`
- `NC_NC_A3_VR`
- `A4_A5_A2_A3`
- `A4_VR_A2_VR`
- `A5_VR_A3_VR`
- `C1_INVERT`
- `C2_INVERT`
- `C1_OUTPUT`
- `C2_OUTPUT`

**Returns:**

`void`

**Function:**

Configures the voltage comparator.

The voltage comparator allows you to compare two voltages and find the greater of them. The configuration constants for this function specify the sources for the comparator in the order C1-C1+, C2-, C2+. The constants may be or’ed together if the NC’s do not overlap; `A4_A5_NC_NC | NC_NC_A3_VR` is valid, however, `A4_A5_NC_NC | A4_VR_NC_NC` may produce unexpected results. The results of the comparator module are stored in C1OUT and C2OUT, respectively. Cx_INVERT will invert the results of the comparator and Cx_OUTPUT will output the results to the comparator output pin.

**Availability:**

Some devices, consult your target datasheet.

**Requires**

Constants are defined in the devices .h file.

**Examples:**

```
setup_comparator(A4_A5_NC_NC); // use C1, not C2
```
setup_compare()

Syntax: setup_compare(x, mode)

Parameters: mode is defined by the constants in the devices .h file
x is 1-16 and specifies which OC pin to use.

Returns: None

Function: This function specifies how the output compare module is going to function based on the value of mode. The device specific options are listed in the device .h file.

Availability: Only available on devices with output compare modules.

Requires: None

Examples:

// Pin OC1 will be set when timer 2 is equal to 0xF000
setup_timer2(TMR_INTERNAL | TIMER_DIV_BY_16);
set_compare_time(1, 0xF000);
setup_compare(1, COMPARE_SET_ON_MATCH | COMPARE_TIMER2);

Also See: set_compare_time(), set_pwm_duty(), setup_capture(), Output Compare / PWM Overview

setup_crc(mode)

Syntax: setup_crc(polyomial terms)

Parameters: polynomial - This will setup the actual polynomial in the CRC engine. The power of each term is passed separated by a comma. 0 is allowed, but ignored. The following define is added to the device's header file (32-bit CRC Moduel Only), to enable little-endian shift direction:

- CRC_LITTLE_ENDIAN

Returns: undefined

Function: Configures the CRC engine register with the polynomial
### Built-in Functions

**Availability:** Only the devices with built in CRC module

**Requires:** Nothing

**Examples:**
```
setup_crc (12, 5);
// CRC Polynomial is X^{12} + X^5 + 1

setup_crc(16, 15, 3, 1);
// CRC Polynomial is X^{16} + X^{15} + X^3 + X^1 + 1
```

**Example Files:** ex.c

**Also See:** `crc_init(); crc_calc(); crc_calc8();`

---

### setup_cog()

**Syntax:**
```
setup_cog(mode, [shutdown]);
setup_cog(mode, [shutdown], [sterring]);
```

**Parameters:**
- **mode** - the setup of the COG module. See the device's .h file for all options. Some typical options include:
  - COG_ENABLED
  - COG_DISABLED
  - COG_CLOCK_HFINTOSC
  - COG_CLOCK_FOSC

- **shutdown** - the setup for the auto-shutdown feature of COG module. See the device's .h file for all the options. Some typical options include:
  - COG_AUTO_RESTART
  - COG_SHUTDOWN_ON_C1OUT
  - COG_SHUTDOWN_ON_C2OUT

- **sterring** - optional parameter for steering the PWM signal to COG output pins and/or selecting the COG pins static level. Used when COG is set for steered PWM or
synchronous steered PWM modes. Not available on all devices, see the device's .h file if available and for all options.
Some typical options include:

- COG_PULSE_STEERING_A
- COG_PULSE_STEERING_B
- COG_PULSE_STEERING_C
- COG_PULSE_STEERING_D

Returns: undefined

Function: Sets up the Complementary Output Generator (COG) module, the auto-shutdown feature of the module and if available steers the signal to the different output pins.

Availability: All devices with a COG module.

Examples: setup_cog(COG_ENABLED | COG_PWM | COG_FALLING_SOURCE_PWM3 | COG_RISING_SOURCE_PWM3, COG_NO_AUTO_SHUTDOWN, COG_PULSE_STEERING_A | COG_PULSE_STEERING_B);

Also See: set_cog_dead_band(), set_cog_phase(), set_cog_blanking(), cog_status(), cog_restart()

**setup_crc( )**

Syntax: setup_crc(polynomia terms)

Parameters: polynomial - This will setup the actual polynomial in the CRC engine. The power of each term is passed separated by a comma. 0 is allowed, but ignored. The following define is added to the device's header file to enable little-endian shift direction: CRC_LITTLE_ENDIAN

Returns: Nothing

Function: Configures the CRC engine register with the polynomial.

Availability: Only devices with a built-in CRC module.

Examples: setup_crc(12, 5); // CRC Polynomial is x^{12}+x^5+1
```c
setup_crc(16, 15, 3, 1); // CRC Polynomial is x^16+x^15+x^3+x^1+1
```

**Example Files:** None

**Also See:** crc_init(), crc_calc(), crc_calc8()

## setup_cwg()

**Syntax:**

```
setup_cwg(mode, shutdown, dead_time_rising, dead_time_falling)
```

**Parameters:**

- **mode** - the setup of the CWG module. See the device's .h file for all options. Some typical options include:
  - CWG_ENABLED
  - CWG.Disabled
  - CWG_OUTPUT_B
  - CWG_OUTPUT_A

- **shutdown** - the setup for the auto-shutdown feature of CWG module. See the device's .h file for all the options. Some typical options include:
  - CWG_AUTO_RESTART
  - CWG_SHUTDOWN_ON)COMP1
  - CWG_SHUTDOWN_ON_FLT
  - CWG_SHUTDOWN_ON_CLC2

- **dead_time_rising** - value specifying the dead time between A and B on the rising edge. (0-63)

- **dead_time_falling** - value specifying the dead time between A and B on the falling edge. (0-63)

**Returns:** undefined

**Function:** Sets up the CWG module, the auto-shutdown feature of module and the rising and falling dead times of the module.
### Availability:
All devices with a CWG module.

### Examples:
```c
setup_cwg(CWG_ENABLED|CWG_OUTPUT_A|CWG_OUTPUT_B|
          CWG_INPUT_PWM1,CWG_SHUTDOWN_ONFLT,60,30);
```

### Example Files:
None

### Also See:
`cwg_status()`, `cwg_restart()`

---

### `setup_dac()`

#### Syntax:
```c
setup_dac(mode);
setup_dac(mode, divisor);
```

#### Parameters:
- **mode** - The valid options vary depending on the device. See the devices .h file for all options. Some typical options include:
  - DAC_OUTPUT
- **divisor** - Divides the provided clock

#### Returns:
undefined

#### Function:
Configures the DAC including reference voltage. Configures the DAC including channel output and clock speed.

#### Availability:
Only the devices with built in digital to analog converter.

#### Requires:
Constants are defined in the devices .h file.

#### Examples:
```c
setup_dac(DAC_VDD | DAC_OUTPUT);
dac_write(value);
setup_dac(DAC_RIGHT_ON, 5);
```

#### Example Files:
None

#### Also See:
`dac_write()`, `DAC Overview`, See header file for device selected
**setup_dci()**

Syntax:  
```c
setup_dci(configuration, data size, rx config, tx config, sample rate);
```

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>configuration</td>
<td>Specifies the configuration the Data Converter Interface should be initialized into, including the mode of transmission and bus properties. The following constants may be combined (OR’d) for this parameter:</td>
</tr>
<tr>
<td>data size</td>
<td>Specifies the size of frames and words in the transmission:</td>
</tr>
<tr>
<td>rx config</td>
<td>Specifies which words of a given frame the DCI module will receive (commonly used for a multi-channel, shared bus situation):</td>
</tr>
<tr>
<td>tx config</td>
<td>Specifies which words of a given frame the DCI module will transmit on:</td>
</tr>
<tr>
<td>sample rate</td>
<td>The desired number of frames per second that the DCI module should produce. Use a numeric value for this parameter. Keep in mind that not all rates are achievable with a given clock. Consult the device datasheet for more information on selecting an adequate clock.</td>
</tr>
</tbody>
</table>

**Returns:** undefined

**Function:** Configures the DCI module
**Availability:** Only on devices with the DCI peripheral

**Requires:** Constants are defined in the devices .h file.

**Examples:**
```c
dci_initialize((I2S_MODE | DCI_MASTER | DCI_CLOCK_OUTPUT |
SAMPLE_RISING_EDGE | UNDERFLOW_LAST |
MULTI_DEVICE_BUS),
DCI_1WORD_FRAME | DCI_16BIT_WORD |
DCI_2WORD_INTERRUPT,
RECEIVE_SLOT0 | RECEIVE_SLOT1,
TRANSMIT_SLOT0 | TRANSMIT_SLOT1,
44100);
```

**Example Files:** None

**Also See:** DCI Overview, dci_start(), dci_write(), dci_read(), dci_transmit_ready(), dci_data_received()

---

**setup_dedicated_adc( )**

**Syntax:**
```c
setup_dedicated_adc(core, mode);
```

**Parameters:**
- `core` - the dedicated ADC core to setup
- `mode` - the mode to setup the dedicated ADC core in. See the device's .h file all options. Some typical options include:
  - ADC_DEDICATED_CLOCK_DIV_2
  - ADC_DEDICATED_CLOCK_DIV_6
  - ADC_DEDICATED_TAD_MUL_2
  - ADC_DEDICATED_TAD_MUL_3

**Returns:** Undefined

**Function:** Configures one of the dedicated ADC core's clock speed and sample time.

Function should be called after the setup_adc() function.

**Availability:** On the dsPIC33EPxxGSxxx family of devices.

**Requires:** Nothing.

**Examples:**
```c
setup_dedicated_adc(0,ADC_DEDICATED_CLOCK_DIV_2 |
ADC_DEDICATED_TAD_MUL_1025);
```
**setup_dma()**

**Syntax:**

```c
setup_dma(channel, peripheral, mode);
```

**Parameters:**

- **Channel** - The channel used in the DMA transfer
- **Peripheral** - The peripheral that the DMA wishes to talk to.
- **Mode** - This will specify the mode used in the DMA transfer

**Returns:**

`void`

**Function:**

Configures the DMA module to copy data from the specified peripheral to RAM allocated for the DMA channel.

**Availability:**

Devices that have the DMA module.

**Requires:**

Nothing

**Examples:**

```c
setup_dma(2, DMA_IN_SPI1, DMA_BYTE);
// This will setup the DMA channel 1 to talk to
// SPI1 input buffer.
```

**Example Files:**

None

**Also See:**

dma_start(), dma_status()

---

**setup_high_speed_adc()**

**Syntax:**

```c
setup_high_speed_adc(mode);
```

**Parameters:**

- **Mode** – Analog to digital mode. The valid options vary depending on the device. See the devices .h file for all options. Some typical options include:

  - ADC_OFF
  - ADC_CLOCK_DIV_1
<table>
<thead>
<tr>
<th>· ADC_HALT_IDLE – The ADC will not run when PIC is idle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns: Undefined</td>
</tr>
<tr>
<td>Function: Configures the High-Speed ADC clock speed and other High-Speed ADC options including, when the ADC interrupts occurs, the output result format, the conversion order, whether the ADC pair is sampled sequentially or simultaneously, and whether the dedicated sample and hold is continuously sampled or samples when a trigger event occurs.</td>
</tr>
<tr>
<td>Availability: Only on dsPIC33FJxxGSxxx devices.</td>
</tr>
<tr>
<td>Requires: Constants are define in the device .h file.</td>
</tr>
<tr>
<td>Examples: setup_high_speed_adc_pair(0, INDIVIDUAL_SOFTWARE_TRIGGER);</td>
</tr>
<tr>
<td>setup_high_speed_adc(ADC_CLOCK_DIV_4);</td>
</tr>
<tr>
<td>read_high_speed_adc(0, START_AND_READ, result);</td>
</tr>
<tr>
<td>setup_high_speed_adc(ADC_OFF);</td>
</tr>
</tbody>
</table>

**Examples Files:** None

**Also See:** setup_high_speed_adc_pair(), read_high_speed_adc(), high_speed_adc_done()

---

### setup_high_speed_adc_pair()

**Syntax:**

```c
setup_high_speed_adc_pair(pair, mode);
```

**Parameters:**

- **pair** – The High-Speed ADC pair number to setup, valid values are 0 to total number of ADC pairs. 0 sets up ADC pair AN0 and AN1, 1 sets up ADC pair AN2 and AN3, etc.

- **mode** – ADC pair mode. The valid options vary depending on the device. See the devices .h file for all options. Some typical options include:
  - INDIVIDUAL_SOFTWARE_TRIGGER
  - GLOBAL_SOFTWARE_TRIGGER
  - PWM_PRIMARY_SE_TRIGGER
  - PWM_GEN1_PRIMARY_TRIGGER
· PWM_GEN2_PRIMARY_TRIGGER

**Returns:** Undefined

**Function:**
Sets up the analog pins and trigger source for the specified ADC pair. Also sets up whether ADC conversion for the specified pair triggers the common ADC interrupt.

If zero is passed for the second parameter the corresponding analog pins will be set to digital pins.

**Availability:**
Only on dsPIC33FJxxGSxxx devices.

**Requires:**
Constants are define in the device .h file.

**Examples:**

```
ssetup_high_speed_adc_pair(0, INDIVIDUAL_SOFTWARE_TRIGGER);
ssetup_high_speed_adc_pair(1, GLOBAL_SOFTWARE_TRIGGER);
ssetup_high_speed_adc_pair(2, 0) – sets AN4 and AN5 as digital pins.
```

**Example Files:**
None

**Also See:**
ssetup_high_speed_adc(), read_high_speed_adc(), high_speed_adc_done()
**Parameters:**

- **unit** - The High Speed PWM unit to set.
- **settings** - Settings to setup the High Speed PWM Leading-Edge Blanking. The valid options vary depending on the device. See the device's header file for all options. Some typical options include:
  - HSPWM_RE_PWMH_TRIGGERS_LE_BLANKING
  - HSPWM_FE_PWMH_TRIGGERS_LE_BLANKING
  - HSPWM_RE_PWSL_TRIGGERS_LE_BLANKING
  - HSPWM_FE_PWSL_TRIGGERS_LE_BLANKING
  - HSPWM_LE_BLANKING_APPLIED_TO_FAULT_INPUT
  - HSPWM_LE_BLANKING_APPLIED_TO_CURRENT_LIMIT_INPUT

- **delay** - 16-bit constant or variable to specify the leading-edge blanking time.

**Returns:** undefined

**Function:** Sets up the Leading-Edge Blanking and leading-edge blanking time of the High Speed PWM.

**Availability:** Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:** None

**Examples:**

```c
setup_hspwm_blanking(HSPWM_RE_PWMH_TRIGGERS_LE_BLANKING, 10);
```

**Example Files:** None

**Also See:**

- setup_hspwm_unit()
- set_hspwm_phase()
- set_hspwm_duty()
- set_hspwm_event()
- setup_hspwm_blanking()
- set_hspwm_override()
- get_hspwm_capture()
- setup_hspwm_chop_clock()
- setup_hspwm_unit_chop_clock()
- setup_hspwm()
- setup_hspwm_secondary()
**setup_hspwm_chop_clock()**

**Syntax:**
```
setup_hspwm_chop_clock(settings);
```

**Parameters:**
- **settings** - a value from 1 to 1024 to set the chop clock divider. Also one of the following can be or'd with the value:
  - HSPWM_CHOP_CLK_GENERATOR_ENABLED
  - HSPWM_CHOP_CLK_GENERATOR_DISABLED

**Returns:**
Undefined

**Function:**
Setup and High Speed PWM Chop Clock Generator and divisor.

**Availability:**
Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**
None

**Examples:**
```
setup_hspwm_chop_clock(HSPWM_CHOP_CLK_GENERATOR_ENABLED|32);
```

**Example Files:**
None

**Also See:**
- setup_hspwm_unit()
- set_hspwm_phase()
- set_hspwm_duty()
- set_hspwm_event()
- setup_hspwm_blanking()
- setup_hspwm_trigger()
- get_hspwm_capture()
- setup_hspwm_unit_chop_clock()
- setup_hspwm()
- setup_hspwm_secondary()

---

**setup_hspwm_trigger()**

**Syntax:**
```
setup_hspwm_trigger(unit, [start_delay], [divider], [trigger_value], [strigger_value]);
```

**Parameters:**
- **unit** - The High Speed PWM unit to set.

- **start_delay** - Optional value from 0 to 63 specifying thenumber of PWM cycles to wait before generating the first trigger event. For some
devices, one of the following may be optional or'd in with the value:
· HSPWM_COMBINE_PRIMARY_AND_SECONDARY_TRIGGER
· HSPWM_SEPERATE_PRIMARY_AND_SECONDARY_TRIGGER

divider - optional value from 1 to 16 specifying the trigger event divisor.

trigger_value - optional 16-bit value specifying the primary trigger compare time.

stringer_value - optional 16-bit value specifying the secondary trigger compare time. Not available on all devices, see the device datasheet for availability.

Returns: undefined

Function: Sets up the High Speed PWM Trigger event.


Requires: None

Examples: setup_hspwm_trigger(1, 10, 1, 0x2000);

Example Files: None

Also See: setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_trigger(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock(), setup_hspwm(), setup_hspwm_secondary()

setup_hspwm_unit( )

Syntax: setup_hspwm_unit(unit, mode, [dead_time], [alt_dead_time]);
set_hspwm_duty(unit, primary, [secondary]);

Parameters: unit - The High Speed PWM unit to set.
### Built-in Functions

| mode - Mode to setup the High Speed PWM unit in. The valid option vary depending on the device. See the device's header file for all options. Some typical options include: |
| · HSPWM_ENABLE |
| · HSPWM_ENABLE_H |
| · HSPWM_ENABLE_L |
| · HSPWM_COMPLEMENTARY |
| · HSPWM_PUSH_PULL |

| dead_time - Optional 16-bit constant or variable to specify the dead time for this PWM unit, defaults to 0 if not specified. |
| alt_dead_time - Optional 16-bit constant or variable to specify the alternate dead time for this PWM unit, default to 0 if not specified. |

| Returns: undefined |

| Function: Sets up the specified High Speed PWM unit. |


| Requires: Constants are defined in the device’s .h file |

| Examples: setup_hspwm_unit(1,HSPWM_ENABLE|HSPWM_COMPLEMENTARY, 100,100); |

| Example Files: None |

| Also See: setup_hspwm_phase(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_blanking(), setup_hspwm_trigger(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock(), setup_hspwm(), setup_hspwm_secondary() |
**setup_hspwm( )  setup_hspwm_secondary( )**

| Syntax: | setup_hspwm(mode, value);  
|         | setup_hspwm_secondary(mode, value);  
|         | //if available |

| Parameters: | mode - Mode to setup the High Speed PWM module in. The valid options vary depending on the device. See the device's .h file for all options. Some typical options include:  
|             | · HSPWM_ENABLED  
|             | · HSPWM_HALT_WHEN_IDLE  
|             | · HSPWM_CLOCK_DIV_1  
|             | value - 16-bit constant or variable to specify the time bases period. |

| Returns: | undefined |

| Function: | To enable the High Speed PWM module and set up the Primary and Secondary Time base of the module. |

| Availability: | Only on devices with a built-in High Speed PWM module  
|               | (dsPIC33FJxxGSxxx, dsPIC33EPxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices) |

| Requires: | Constants are defined in the device's .h file |

| Examples: | setup_hspwm(HSPWM_ENABLED | HSPWM_CLOCK_DIV_BY4, 0x8000); |

| Example Files: | None |

| Also See: | setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_duty(),  
|           | set_hspwm_event(),  
|           | setup_hspwm_blanking(), setup_hspwm_trigger(),  
|           | set_hspwm_override(),  
|           | get_hspwm_capture(), setup_hspwm_chop_clock(),  
|           | setup_hspwm_unit_chop_clock()  
|           | setup_hspwm_secondary() |
**setup_hspwm_unit_chop_clock( )**

**Syntax:**
```
setup_hspwm_unit_chop_clock(unit, settings);
```

**Parameters:**
- `unit` - the High Speed PWM unit chop clock to setup.
- `settings` - a settings to setup the High Speed PWM unit chop clock. The valid options vary depending on the device. See the device's .h file for all options. Some typical options include:
  - HSPWM_PWMH_CHOPPING_ENABLED
  - HSPWM_PWML_CHOPPING_ENABLED
  - HSPWM_CHOPPING_DISABLED
  - HSPWM_CLOP_CLK_SOURCE_PWM2H
  - HSPWM_CLOP_CLK_SOURCE_PWM1H
  - HSPWM_CHOP_CLK_SOURCE_CHOP_CLK_GENERATOR

**Returns:** Undefined

**Function:** Setup and High Speed PWM unit's Chop Clock

**Availability:** Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:** None

**Examples:**
```
setup_hspwm_unit_chop_clock(1, HSPWM_PWMH_CHOPPING_ENABLED | HSPWM_PWML_CHOPPING_ENABLED | HSPWM_CLOP_CLK_SOURCE_PWM2H);
```

**Example Files:** None

**Also See:**
- `setup_hspwm_unit()`, `set_hspwm_phase()`, `set_hspwm_duty()`, `set_hspwm_event()`, `setup_hspwm_blanking()`, `setup_hspwm_trigger()`, `set_hspwm_override()`, `get_hspwm_capture()`, `setup_hspwm_chop_clock()`, `setup_hspwm()`, `setup_hspwm_secondary()`
setup_low_volt_detect( )

Syntax: setup_low_volt_detect(mode)

Parameters: mode may be one of the constants defined in the devices .h file. LVD_LVDIN, LVD_45, LVD_42, LVD_40, LVD_38, LVD_36, LVD_35, LVD_33, LVD_30, LVD_28, LVD_27, LVD_25, LVD_23, LVD_21, LVD_19 One of the following may be or’ed(via |) with the above if high voltage detect is also available in the device LVD_TRIGGER_BELOW, LVD_TRIGGER_ABOVE

Returns: undefined

Function: This function controls the high/low voltage detect module in the device. The mode constants specifies the voltage trip point and a direction of change from that point (available only if high voltage detect module is included in the device). If the device experiences a change past the trip point in the specified direction the interrupt flag is set and if the interrupt is enabled the execution branches to the interrupt service routine.

Availability: This function is only available with devices that have the high/low voltage detect module.

Requires Constants are defined in the devices.h file.

Examples: setup_low_volt_detect( LVD_TRIGGER_BELOW | LVD_36 );

This would trigger the interrupt when the voltage is below 3.6 volts

setup_motor_pwm( )

Syntax: setup_motor_pwm(pwm,options, timebase);
setup_motor_pwm(pwm,options,prescale,postscale,timebase)

Parameters: Pwm- Defines the pwm module used.

Options- The mode of the power PWM module. See the devices .h file for all options

timebase- This parameter sets up the PWM time base pre-scale and
Built-in Functions

**setup_motor_pwm()**

**Syntax:**

```c
setup_motor_pwm(1, MPWM_FREE_RUN | MPWM_SYNC_OVERRIDES, timebase);
```

**Parameters:**

- Mode is one of:
  - OSC_INTERNAL
  - OSC_CRYSTAL
  - OSC_CLOCK
  - OSC_RC
  - OSC_SECONDARY

- Target is the target frequency to run the device it.

- Source is optional. It specifies the external crystal/oscillator frequency. If omitted the value from the last #USE_DELAY is used. If mode is OSCINTERNAL, source is an optional tune value for the internal oscillator for PICs that support it. If omitted a tune value of zero will be used.

**Returns:**

`void`

**Function:** Configures the motor control PWM module

**Availability:** Devices that have the motor control PWM unit.

**Requires:** None

**Examples:**

```c
setup_motor_pwm(1, MPWM_FREE_RUN | MPWM_SYNC_OVERRIDES, timebase);
```

**Also See:**

- `get_mOTOR_pwm_count()`
- `set_mOTOR_pwm_event()`
- `set_mOTOR_unit()`
- `set_mOTOR_pwm_duty()`

---

**setup_oscillator()**

**Syntax:**

```c
setup_oscillator(mode, target [source] [divide])
```

**Parameters:**

- Mode is one of:
  - OSC_INTERNAL
  - OSC_CRYSTAL
  - OSC_CLOCK
  - OSC_RC
  - OSC_SECONDARY

- Target is the target frequency to run the device it.

- Source is optional. It specifies the external crystal/oscillator frequency. If omitted the value from the last #USE_DELAY is used. If mode is OSC_INTERNAL, source is an optional tune value for the internal oscillator for PICs that support it. If omitted a tune value of zero will be used.

---

421
Divide in optional. For PICs that support it, it specifies the divide ration for
the Display Module Interface Clock. A number from 0 to 64 divides the
clock from 1 to 17 increasing in increments of 0.25, a number from 64 to
96 divides the clock from 17 to 33 increasing in increments of 0.5, and a
number from 96 to 127 divides the clock from 33 to 64 increasing in
increments of 1. If omitted zero will be used for divide by 1.

Returns: None

Function: Configures the oscillator with preset internal and external source
configurations. If the device fuses are set and #use delay() is specified,
the compiler will configure the oscillator. Use this function for explicit
configuration or programming dynamic clock switches. Please consult
your target data sheets for valid configurations, especially when using the
PLL multiplier, as many frequency range restrictions are specified.

Availability: This function is available on all devices.

Requires: The configuration constants are defined in the device’s header file.

Examples: \n\nsetup_oscillator( OSC_CRYSTAL, 4000000, 16000000); \nsetup_oscillator( OSC_INTERNAL, 29480000);

Example Files: None

Also See: setup_wdt(), Internal Oscillator Overview

---

**setup_pga()**

Syntax: \n\nsetup_pga(module,settings)

Parameters: \n\nmodule - constant specifying the Programmable Gain Amplifier (PGA) to
setup.

Returns: Undefined

Function: This function allows for setting up one of the Programmable Gain
Amplifier modules.

Availability: Devices with a Programmable Gain Amplifier module.
Built-in Functions

Requires: Nothing.

Examples: setup_pga(PGA_ENABLED | PGA_POS_INPUT_PGAxP1 | PGA_GAIN_8X);

Example Files: None

Also See:

---

**setup_pid( )**

Syntax: `setup_pid([mode],[K1],[K2],[K3]);`

Parameters: `mode`- the setup of the PID module. The options for setting up the module are defined in the device's header file as:
- PID_MODE_PID
- PID_MODE_SIGNED_ADD_MULTIPLY_WITH_ACCUMULATION
- PID_MODE_SIGNED_ADD_MULTIPLY
- PID_MODE_UNSIGNED_ADD_MULTIPLY_WITH_ACCUMULATION
- PID_MODE_UNSIGNED_ADD_MULTIPLY
- PID_OUTPUT_LEFT_JUSTIFIED
- PID_OUTPUT_RIGHT_JUSTIFIED

- **K1** - optional parameter specifying the K1 coefficient, defaults to zero if not specified. The K1 coefficient is used in the PID and ADD_MULTIPLY modes. When in PID mode the K1 coefficient can be calculated with the following formula:
  - K1 = Kp + Ki * T + Kd/T

  When in one of the ADD_MULTIPLY modes K1 is the multiple value.

- **K2** - optional parameter specifying the K2 coefficient, defaults to zero if not specified. The K2 coefficient is used in the PID mode only and is calculated with the following formula:
  - K2 = -(Kp + 2Kd/T)

- **K3** - optional parameter specifying the K3 coefficient, defaults to zero if not specified. The K3 coefficient is used in the PID mode, only and is calculated with the following formula:
  - K3 = Kd/T

  T is the sampling period in the above formulas.

Returns: Nothing
Function: To setup the Proportional Integral Derivative (PID) module, and to set the input coefficients (K1, K2 and K3).

Availability: All devices with a PID module.

Requires: Constants are defined in the device's .h file.

Examples: 

```c
setup_pid(PID_MODE_PID, 10, -3, 50);
```

Example Files: None

Also See: `pid_get_result()`, `pid_read()`, `pid_write()`, `pid_busy()`

---

**setup_pmp(option,address_mask)**

Syntax: `setup_pmp(option,address_mask);`

Parameters:  
- **options**- The mode of the Parallel Master Port that allows to set the Master Port mode, read-write strobe options and other functionality of the PMPort module. See the device's .h file for all options. Some typical options include:
  - PAR_PSP_AUTO_INC
  - PAR_CONTINUE_IN_IDLE
  - PAR_INTR_ON_RW //Interrupt on read write
  - PAR_INC_ADDR //Increment address by 1 every
  - PAR_MASTER_MODE_1 //Master Mode 1
  - PAR_WAITE4 //4 Tcy Wait for data hold after

- **address_mask**- this allows the user to setup the address enable register with a 16-bit value. This value determines which address lines are active from the available 16 address lines PMA0:PMA15.

Returns: Undefined.

Function: Configures various options in the PMP module. The options are present in
the device's .h file and they are used to setup the module. The PMP module is highly configurable and this function allows users to setup configurations like the Slave module, Interrupt options, address increment/decrement options, Address enable bits, and various strobe and delay options.

**Availability:** Only the devices with a built-in Parallel Master Port module.

**Requires:** Constants are defined in the device's .h file.

**Examples:**
```c
setup_psp(PAR_ENABLE|
         PAR_MASTER_MODE_1|PAR_
         STOP_IN_IDLE,0x00FF);
```

**Example Files:** None

**Also See:**
- setup_pmp( ), pmp_address( ), pmp_read( ), psp_read( ), psp_write( ),
- pmp_write( ), psp_output_full( ), psp_input_full( ), psp_overflow( ),
- pmp_output_full( ), pmp_input_full( ), pmp_overflow()

See header file for device selected

## setup_psmc( )

**Syntax:**
```c
setup_psmc(unit, mode, period, period_time, rising_edge, rise_time, falling_edge, fall_time);
```

**Parameters:**
- **unit** is the PSMC unit number 1-4
- **mode** is one of:
  - PSMC_SINGLE
  - PSMC_PUSH_PULL
  - PSMC_BRIDGE_PUSH_PULL
  - PSMC_PULSE.SKIPPING
  - PSMC_ECCP_BRIDGE_REVERSE
  - PSMC_ECCP_BRIDGE_FORWARD
  - PSMC_VARIABLE_FREQ
  - PSMC_3_PHASE

For complementary outputs use a or bar (|) and PSMC_COMPLEMENTARY
Normally the module is not started until the `psmc_pins()` call is made. To enable immediately or in `PSMC_ENABLE_NOW`.

**period** has three parts or'ed together. The clock source, the clock divisor and the events that can cause the period to start.

Sources:
- `PSMC_SOURCE_FOSC`
- `PSMC_SOURCE_64MHZ`
- `PSMC_SOURCE_CLK_PIN`

Divisors:
- `PSMC_DIV_1`
- `PSMC_DIV_2`
- `PSMC_DIV_4`
- `PSMC_DIV_8`

Events:
- Use any of the events listed below.

**period_time** is the duration the period lasts in ticks. A tick is the above clock source divided by the divisor.

**rising_edge** is any of the following events to trigger when the signal goes active.

**rise_time** is the time in ticks that the signal goes active (after the start of the period) if the event is `PSMC_EVENT_TIME`, otherwise unused.

**falling_edge** is any of the following events to trigger when the signal goes inactive.

**fall_time** is the time in ticks that the signal goes inactive (after the start of the period) if the event is `PSMC_EVENT_TIME`, otherwise unused.

Events:
- `PSMC_EVENT_TIME`
- `PSMC_EVENT_C1OUT`
- `PSMC_EVENT_C2OUT`
- `PSMC_EVENT_C3OUT`
- `PSMC_EVENT_C4OUT`
- `PSMC_EVENT_PIN_PIN`
### setup_psmc()

**Syntax:**
```
setup_psmc(module, PSMC_TYPE, PSMC_FIXED | PSMC_EVENT_TIME, us(value), PSMC_FIXED | PSMC_EVENT_TIME, us(value));
```

**Parameters:**
- `module`: Integer representing the module number.
- `PSMC_TYPE`: Integer representing the PSMC type.
- `PSMC_FIXED`: Boolean indicating fixed settings.
- `PSMC_EVENT_TIME`: Boolean indicating event time.
- `us(value)`: Integer representing the value in microseconds.

**Returns:**
`undefined`

**Function:**
Initializes a PSMC unit with the primary characteristics such as the type of PWM, the period, duty and various advanced triggers. Normally this call does not start the PSMC. It is expected all the setup functions be called and the psmc_pins() be called last to start the PSMC module. These two calls are all that are required for a simple PWM. The other functions may be used for advanced settings and to dynamically change the signal.

**Availability:**
All devices equipped with PSMC module.

**Requires:**
None

**Examples:**
```c
setup_psmc(1, PSMC_SINGLE, PSMC_EVENT_TIME | PSMC_SOURCE_FOSC, us(100), PSMC_EVENT_TIME, 0, PSMC_EVENT_TIME, us(25));
psmc_pins(1, PSMC_A);
```

### setup_power_pwm_pins()

**Syntax:**
```
setup_power_pwm_pins(module0, module1, module2, module3)
```

**Parameters:**
- `module0`, `module1`, `module2`, `module3`: Integers representing the module numbers.

**Returns:**
`undefined`

**Function:**
Configures the pins of the Pulse Width Modulation (PWM) device.

**Availability:**
All devices equipped with a power control PWM.

**Requires:**
None
Examples:

```
setup_power_pwm_pins(PWM_PINS_DISABLED, PWM_PINS_DISABLED,
PWM_PINS_DISABLED);
setup_power_pwm_pins(PWM_COMPLEMENTARY, PWM_COMPLEMENTARY, PWM_PINS_DISABLED, PWM_PINS_DISABLED);
```

Example

Files: None

Also See:

setup_power_pwm(),
set_power_pwm_override(), set_power_pwmX_duty()

---

**setup_psp(option,address_mask)**

**Syntax:**

```
setup_psp (options.address_mask);
setup_psp (options);
```

**Parameters:**

- **Option:** The mode of the Parallel slave port. This allows to set the slave port mode, read-write strobe options and other functionality of the PMP/EPMP module. See the devices.h file for all options. Some typical options include:
  - PAR_PSP_AUTO_INC
  - PAR_CONTINUE_IN_IDLE
  - PAR_INTR_ON_RW //Interrupt on read write
  - PAR_INC_ADDR //Increment address by 1 every
  - PAR_WAITE4 //4 Tcy Wait for data hold after

- **address_mask:** This allows the user to setup the address enable register with a 16 bit or 32 bit (EPMP) value. This value determines which address lines are active from the available 16 address lines PMA0: PMA15 or 32 address lines PMAO:PMA31 (EPMP only).

**Returns:** Undefined.

**Function:** Configures various options in the PMP/EPMP module. The options are present in the device.h file and they are used to setup the module. The
PMP/EPMP module is highly configurable and this function allows users to setup configurations like the Slave mode, Interrupt options, address increment/decrement options, Address enable bits and various strobe and delay options.

**Availability:** Only the devices with a built in Parallel Port module or Enhanced Parallel Master Port module.

**Requires:** Constants are defined in the devices .h file.

**Examples:**
```
setup_psp(PAR_PSP_AUTO_INC|PAR_STOP_IN_IDLE,0x00FF );
```
//Sets up legacy slave mode with read and write buffers auto increment.

**Example Files:** None

**Also See:**
- setup_pmp()
- pmp_address()
- pmp_read()
- psp_read()
- psp_write()
- pmp_write()
- psp_output_full()
- psp_input_full()
- psp_overflow()
- pmp_output_full()
- pmp_input_full()
- pmp_overflow()

See header file for device selected.

### setup_pwm1() setup_pwm2() setup_pwm3() setup_pwm4()

**Syntax:**
```
setup_pwm1(settings);
setup_pwm2(settings);
setup_pwm3(settings);
setup_pwm4(settings);
```

**Parameters:**
- **settings** - setup of the PWM module. See the device's .h file for all options.
  - Some typical options include:
    - PWM_ENABLED
    - PWM_OUTPUT
    - PWM_ACTIVE_LOW

**Returns:** Undefined
**Function:**
Sets up the PWM module.

**Availability:**
On devices with a PWM module.

**Examples:**
```c
setup_pwm1(PWM_ENABLED|PWM_OUTPUT);
```

**Example Files:**
None

**Also See:**
`set_pwm_duty()`

## setup_qei()

**Syntax:**
```c
setup_qei( [unit,]options, filter, maxcount );
```

**Parameters:**
- **Options** - The mode of the QEI module. See the devices .h file for all options
  
  Some common options are:
  - QEI_MODE_X2
  - QEI_TIMER_GATED
  - QEI_TIMER_DIV_BY_1

- **filter** - This parameter is optional and the user can specify the digital filter clock divisor.

- **maxcount** - This will specify the value at which to reset the position counter.

- **unit** - Optional unit number, defaults to 1.

**Returns:**
`void`

**Function:**
Configures the Quadrature Encoder Interface. Various settings like modes, direction can be setup.

**Availability:**
Devices that have the QEI module.

**Requires:**
Nothing.

**Examples:**
```c
setup_qei(QEI_MODE_X2|QEI_TIMER_INTERNAL,QEI_FILTER_DIV_2,
QEI_FORWARD);
```
setup_rtc( )

Syntax: `setup_rtc() (options, calibration);`

Parameters:
- **Options**: The mode of the RTCC module. See the devices .h file for all options
- **Calibration**: This parameter is optional and the user can specify an 8 bit value that will get written to the calibration configuration register.

Returns: `void`

Function: Configures the Real Time Clock and Calendar module. The module requires an external 32.768 kHz clock crystal for operation.

Availability: Devices that have the RTCC module.

Requires: Nothing.

Examples: `setup_rtc(RTC_ENABLE | RTC_OUTPUT SECONDS, 0x00);`
// Enable RTCC module with seconds clock and no calibration

Example Files: None

Also See: `rtc_read(), rtc_alarm_read(), rtc_alarm_write(), setup_rtc_alarm(), rtc_write(), setup_rtc()`

setup_rtc_alarm( )

Syntax: `setup_rtc_alarm(options, mask, repeat);`

Parameters:
- **options**: The mode of the RTCC module. See the devices .h file for all options
- **mask**: specifies the alarm mask bits for the alarm configuration.
repeat - Specifies the number of times the alarm will repeat. It can have a max value of 255.

Returns: void

Function: Configures the alarm of the RTCC module.

Availability: Devices that have the RTCC module.

Requires: Nothing.

Examples: setup_rtc_alarm(RTC_ALARM_ENABLE, RTC_ALARM_HOUR, 3);

Example Files: None

Also See: rtc_read(), rtc_alarm_read(), rtc_alarm_write(), setup_rtc_alarm(), rtc_write(), setup_rtc()

setup_sd_adc()

Syntax: setup_sd_adc(settings1, settings2, settings3);

Parameters: settings1 - settings for the SD1CON1 register of the SD ADC module. See the device's .h file for all options. Some options include:
1  SDADC_ENABLED
2  SDADC_NO_HALT
3  SDADC_GAIN_1
4  SDADC_NO_DITHER
5  SDADC_SVDD_SVSS
6  SDADC_BW_NORMAL

settings2 - settings for the SD1CON2 register of the SD ADC module. See the device's .h file for all options. Some options include:
7  SDADC_CHOPPING_ENABLED
8  SDADC_INT_EVERY_SAMPLE
9  SDADC_RES_UPDATED_EVERY_INT
10  SDADC_NO_ROUNDING

settings3 - settings for the SD1CON3 register of the SD ADC module. See the device's .h file for all options. Some options include:
11  SDADC_CLOCK_DIV_1
Built-in Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup_sd_adc(SDADC_ENABLED</td>
<td>SDADC_DITHER_LOW, SDADC_CHOPPING_ENABLED</td>
<td>SDADC_INT_EVERY_5TH_SAMPLE</td>
</tr>
</tbody>
</table>

Example Files: None

Also See: set_sd_adc_channel(), read_sd_adc(), set_sd_adc_calibration()

setup_smt1( )

Syntax: setup_smt1(mode,[period]);
set up_smt2(mode,[period]);

Parameters: mode - The setup of the SMT module. See the device's .h file for all options. Some typical options include:
- SMT_ENABLED
- SMT_MODE_TIMER
- SMT_MODE_GATED_TIMER
- SMT_MODE_PERIOD_DUTY_CYCLE_ACQ

period - Optional parameter for specifying the overflow value of the SMT timer, defaults to maximum value if not specified.

Returns: Nothing
Function: Configures the Signal Measurement Timer (SMT) module.

Availability: Only devices with a built-in SMT module.

Examples:
```c
setup_smt1(SMT_ENABLED | SMT_MODE_PERIOD_DUTY_CYCLE_ACQ |
SMT_REPEAT_DATA_ACQ_MODE | SMT_CLK_FOSC);
```

Example Files: None

Also See: `smtx_status()`, `stmx_start()`, `smtx_stop()`, `smtx_update()`, `smtx_reset_timer()`, `smtx_read()`, `smtx_write()`

---

**setup_spi( )  setup_spi2( )**

Syntax:
```c
setup_spi (mode)
setup_spi2 (mode)
```

Parameters: `mode` may be:
- SPI_MASTER, SPI_SLAVE, SPI_SS_DISABLED
- SPI_L_TO_H, SPI_H_TO_L
- SPI_CLK_DIV_4, SPI_CLK_DIV_16,
- SPI_CLK_DIV_64, SPI_CLK_T2
- SPI_SAMPLE_AT_END, SPI_XMIT_L_TO_H
- SPI_MODE_16B, SPI_XMIT_L_TO_H
- Constants from each group may be or'ed together with
  `|`.

Returns: undefined

Function: Configures the hardware SPI™ module.
- SPI_MASTER will configure the module as the bus master
- SPI_SLAVE will configure the module as a slave on the SPI™ bus
- SPI_SS_DISABLED will turn off the slave select pin so the slave module
  receives any transmission on the bus.
- SPI_x_to_y will specify the clock edge on which to sample and transmit
  data
- SPI_CLK_DIV_x will specify the divisor used to create the SCK clock
  from system clock.

Availability: This function is only available on devices with SPI hardware.

Requires: Constants are defined in the devices .h file.
setup_timerx( )

Syntax: setup_timerX(mode)
setup_timerX(mode,period)

Parameters: Mode is a bit-field comprised of the following configuration constants:

- TMR_DISABLED: Disables the timer operation.
- TMR_INTERNAL: Enables the timer operation using the system clock. Without divisions, the timer will increment on every instruction cycle. On PCD, this is half the oscillator frequency.
- TMR_EXTERNAL: Uses a clock source that is connected to the SOSCI/SOSCO pins
- TMR_EXTERNAL_SYNC: Uses a clock source that is connected to the SOSCI/SOSCO pins. The timer will increment on the rising edge of the external clock which is synchronized to the internal clock phases. This mode is available only for Timer1.
- TMR_EXTERNAL_RTC: Uses a low power clock source connected to the SOSCI/SOSCO pins; suitable for use as a real time clock. If this mode is used, the low power oscillator will be enabled by the setup_timer function. This mode is available only for Timer1.
- TMR_DIV_BY_X: X is the number of input clock cycles to pass before the timer is incremented. X may be 1, 8, 64 or 256.
- TMR_32_BIT: This configuration concatenates the timers into 32 bit mode. This constant should be used with timers 2, 4, 6 and 8 only.
- Period is an optional 16 bit integer parameter that specifies the timer

Examples: setup_spi(SPI_MASTER | SPI_L_TO_H | SPI_DIV_BY_16);

Example Files: ex_spi.c
Also See: spi_write(), spi_read(), spi_data_is_in(), SPI Overview
period. The default value is 0xFFFF.

Returns: void

Function: Sets up the timer specified by X (May be 1 – 9). X must be a valid timer on the target device.

Availability: This function is available on all devices that have a valid timer X. Use getenv or refer to the target datasheet to determine which timers are valid.

Requires: Configuration constants are defined in the device's header file.

Examples: /* setup a timer that increments every 64th instruction cycle with an overflow period of 0xA010 */
setup_timer2(TMR_INTERNAL | TMR_DIV_BY_64, 0xA010);

/* Setup another timer as a 32-bit hybrid with a period of 0xFFFFFFFF and a interrupt that will be fired when that timer overflows*/
setup_timer4(TMR_32_BIT); //use get_timer45() to get the timer value
enable_interrups(int_timer5); //use the odd number timer for the interrupt

Example Files: None

Also See: Timer Overview, setup_timerX(), get_timerXY(), set_timerX(), set_timerXY()
### setup_timer_B()

**Syntax:**

```
setup_timer_B (mode);
```

**Parameters:**

- `mode` values may be:
  - `TB_OFF`, `TB_INTERNAL`, `TB_EXT_H_TO_L`, `TB_EXT_L_TO_H`
  - `TB_DIV_1`, `TB_DIV_2`, `TB_DIV_4`, `TB_DIV_8`, `TB_DIV_16`, `TB_DIV_32`, `TB_DIV_64`, `TB_DIV_128`, `TB_DIV_256`
  - constants from different groups may be or'ed together with `|`.

**Returns:**

undefined

**Function:**

sets up Timer B

**Availability:**

This function is only available on devices with Timer B hardware.

**Requires:**

Constants are defined in device's .h file.

**Examples:**

```
ssetup_timer_B(TB_OFF);
ssetup_timer_B(TB_INTERNAL | TB_DIV_256);
ssetup_timer_B(TA_EXT_L_TO_H | TA_DIV_1);
```

**Example Files:**

none

**Also See:**

`get_timerB()`, `set_timerB()`, TimerB Overview
### setup_timer_0()

**Syntax:**

```c
setup_timer_0 (mode)
```

**Parameters:**
`mode` may be one or two of the constants defined in the devices .h file. RTCC_INTERNAL, RTCC_EXT_L_TO_H or RTCC_EXT_H_TO_L

- RTCC_DIV_2, RTCC_DIV_4, RTCC_DIV_8, RTCC_DIV_16,
- RTCC_DIV_32, RTCC_DIV_64, RTCC_DIV_128, RTCC_DIV_256

PIC18XXX only: RTCC_OFF, RTCC_8_BIT

One constant may be used from each group or’ed together with the `|` operator.

**Returns:**
undefined

**Function:**
Sets up the timer 0 (aka RTCC).

**Warning:**
On older PIC16 devices, set-up of the prescaler may undo the WDT prescaler.

**Availability:**
All devices.

**Requires:**
Constants are defined in the devices .h file.

**Examples:**

```
setup_timer_0 (RTCC_DIV_2 | RTCC_EXT_L_TO_H);
```

**Example Files:**
Also See: `get_timer0()`, `set_timer0()`, `setup counters()`

### setup_timer_1()

**Syntax:**

```c
setup_timer_1 (mode)
```

**Parameters:**
`mode` values may be:

- `T1DISABLED`, `T1_INTERNAL`, `T1_EXTERNAL`,
  `T1_EXTERNAL_CAPTURE`, `T1_EXTERNAL_COMPARE`
Built-in Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Initializes timer 1. The timer value may be read and written to using SET_TIMER1() and GET_TIMER1(). Timer 1 is a 16 bit timer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>This function is only available on devices with timer 1 hardware.</td>
</tr>
<tr>
<td>Requires</td>
<td>Constants are defined in the devices .h file.</td>
</tr>
<tr>
<td>Examples</td>
<td>setup_timer_1 ( T1_DISABLED ); setup_timer_1 ( T1_INTERNAL</td>
</tr>
<tr>
<td>Also See</td>
<td>get_timer1(), Timer1 Overview</td>
</tr>
</tbody>
</table>

**setup_timer_2( )**

**Syntax:** setup_timer_2 (mode, period, postscale)

**Parameters:** mode may be one of:
- T2_DISABLED
- T2_DIV_BY_1, T2_DIV_BY_4, T2_DIV_BY_16

Period is a int 0-255 that determines when the clock value is reset
Postscale is a number 1-16 that determines how many timer overflows before an interrupt: (1 means once, 2 means twice, an so on)

**Returns:** undefined
Function: Initializes timer 2. The mode specifies the clock divisor (from the oscillator clock). The timer value may be read and written to using GET_TIMER2() and SET_TIMER2().
2 is a 8-bit counter/timer.

Availability: This function is only available on devices with timer 2 hardware.

Requires: Constants are defined in the devices .h file.

Examples:
```
setup_timer_2 ( T2_DIV_BY_4, 0xc0, 2) //at 20mhz, the timer
will
//increment every
800ns
//will overflow every
154.4us,
//and will interrupt
every 308.us
```

Example Files:

Also See: get_timer2(), set_timer2() Timer2 Overview

---

**setup_timer_3( )**

Syntax: `setup_timer_3 (mode)`

Parameters: **Mode** may be one of the following constants from each group or’ed (via |) together:
- T3_DISABLED, T3_INTERNAL, T3_EXTERNAL, T3_EXTERNAL_SYNC
- T3_DIV_BY_1, T3_DIV_BY_2, T3_DIV_BY_4, T3_DIV_BY_8

Returns: undefined

Function: Initializes timer 3 or 4. The mode specifies the clock divisor (from the oscillator clock). The timer value may be read and written to using GET_TIMER3() and SET_TIMER3(). Timer 3 is a 16 bit counter/timer.
**Built-in Functions**

**Availability:** This function is only available on devices with timer 3 hardware.

**Requires:** Constants are defined in the devices .h file.

**Examples:**

```
setup_timer_3 (T3_INTERNAL | T3_DIV_BY_2);
```

**Example Files:** None

**Also See:** get_timer3(), set_timer3()

### setup_timer_4()

**Syntax:**

```
setup_timer_4 (mode, period, postscale)
```

**Parameters:**

- **mode** may be one of:
  - T4_DISABLED, T4_DIV_BY_1, T4_DIV_BY_4, T4_DIV_BY_16

- **period** is a int 0-255 that determines when the clock value is reset,

- **postscale** is a number 1-16 that determines how many timer overflows before an interrupt: (1 means once, 2 means twice, and so on).

**Returns:** undefined

**Function:**

Initializes timer 4. The mode specifies the clock divisor (from the oscillator clock).
The timer value may be read and written to using GET_TIMER4() and SET_TIMER4().
Timer 4 is a 8 bit counter/timer.

**Availability:** This function is only available on devices with timer 4 hardware.

**Requires:** Constants are defined in the devices .h file

**Examples:**

```
setup_timer_4 ( T4_DIV_BY_4, 0xc0, 2);
// At 20mhz, the timer will increment every 800ns,
// will overflow every 153.6us,
// and will interrupt every 307.2us.
```
**setup_timer_5()**

**Syntax:**

```
setup_timer_5 (mode)
```

**Parameters:**

- `mode` may be one or two of the constants defined in the devices .h file.

  - T5_DISABLED, T5_INTERNAL, T5_EXTERNAL, or T5_EXTERNAL_SYNC
  - T5_DIV_BY_1, T5_DIV_BY_2, T5_DIV_BY_4, T5_DIV_BY_8
  - T5_ONE_SHOT, T5_DISABLE_SE_RESET, or T5_ENABLE_DURING_SLEEP

**Returns:**

undefined

**Function:**

Initializes timer 5. The mode specifies the clock divisor (from the oscillator clock). The timer value may be read and written to using GET_TIMER5() and SET_TIMER5(). Timer 5 is a 16 bit counter/timer.

**Availability:**

This function is only available on devices with timer 5 hardware.

**Requires:**

Constants are defined in the devices .h file.

**Examples:**

```
setup_timer_5 (T5_INTERNAL | T5_DIV_BY_2);
```

**Example Files:**

None

**Also See:**

get_timer5(), set_timer5(), Timer5 Overview
**setup_uart( )**

**Syntax:**

- `setup_uart(baud, stream)`
- `setup_uart(baud)`
- `setup_uart(baud, stream, clock)`

**Parameters:**

- `baud` is a constant representing the number of bits per second. A one or zero may also be passed to control the on/off status.
- `Stream` is an optional stream identifier.

Chips with the advanced UART may also use the following constants:
- UART_ADDRESS: UART only accepts data with 9th bit=1
- UART_DATA: UART accepts all data

Chips with the EUART H/W may use the following constants:
- UART_AUTODETECT: Waits for 0x55 character and sets the UART baud rate to match.
- UART_AUTODETECT_NOWAIT: Same as above function, except returns before 0x55 is received. KBHIT() will be true when the match is made. A call to GETC() will clear the character.
- UART_WAKEUP_ON_RDA: Wakes PIC up out of sleep when RCV goes from high to low

- `clock` - If specified this is the clock rate this function should assume. The default comes from the #USE DELAY.

**Returns:**

undefined

**Function:**

Very similar to SET_UART_SPEED. If 1 is passed as a parameter, the UART is turned on, and if 0 is passed, UART is turned off. If a BAUD rate is passed to it, the UART is also turned on, if not already on.

**Availability:**

This function is only available on devices with a built in UART.

**Requires:**

#USE RS232

**Examples:**

- `setup_uart(9600);`
- `setup_uart(9600, rsOut);`

**Example Files:**

None

**Also See:**

#USE RS232, putc(), getc(), RS232 I/O Overview
### setup_vref( )

**Syntax:**
```
setup_vref (mode)
```

**Parameters:**
- `mode` is a bit-field comprised of the following constants:
  - VREF_DISABLED
  - VREF_LOW ( Vdd * value / 24)
  - VREF_HIGH ( Vdd * value / 32 + Vdd/4 )
  - VREF_ANALOG

**Returns:** undefined

**Function:** Configures the voltage reference circuit used by the voltage comparator.

The voltage reference circuit allows you to specify a reference voltage that the comparator module may use. You may use the Vdd and Vss voltages as your reference or you may specify VREF_ANALOG to use supplied Vdd and Vss. Voltages may also be tuned to specific values in steps, 0 through 15. That value must be or’ed to the configuration constants.

**Availability:** Some devices, consult your target datasheet.

**Requires:** Constants are defined in the devices .h file.

**Examples:**
```
/* Use the 15th step on the course setting */
setup_vref(VREF_LOW | 14);
```

**Example Files:** None

### setup_wdt( )

**Syntax:**
```
setup_wdt (mode)
```

**Parameters:** Mode is a bit-field comprised of the following constants:
- WDT_ON
- WDT_OFF

Specific Time Options vary between chips, some examples are:
```
WDT_2ms
```
**Built-in Functions**

| Function | Configures the watchdog timer. The watchdog timer is used to monitor the software. If the software does not reset the watchdog timer before it overflows, the device is reset, preventing the device from hanging until a manual reset is initiated. The watchdog timer is derived from the slow internal timer. |

**Availability:**

**Examples:**

```c
setup_wdt(WDT_ON);
```

**Example Files:** ex_wdt.c

**Also See:** [Internal Oscillator Overview](#)

---

**setup_zdc( )**

**Syntax:**

```c
setup_zdc(mode);
```

**Parameters:**

- **mode**: the setup of the ZDC module. The options for setting up the module include:
  - ZCD_ENABLED
  - ZCD_DISABLED
  - ZCD_INVERTED
  - ZCD_INT_L_TO_H
  - ZCD_INT_H_TO_L

**Returns:** Nothing

**Function:** To set-up the Zero_Cross Detection (ZCD) module.

**Availability:** All devices with a ZCD module.

**Examples:**

```c
setup_zcd(ZCD_ENABLE|ZCD_INT_H_TO_L);
```

**Example Files:** None

**Also See:** zcd_status()
# shift_left( )

**Syntax:**

```plaintext
shift_left (address, bytes, value)
```

**Parameters:**
- `address` is a pointer to memory.
- `bytes` is a count of the number of bytes to work with.
- `value` is a 0 to 1 to be shifted in.

**Returns:**
0 or 1 for the bit shifted out

**Function:**
Shifts a bit into an array or structure. The address may be an array identifier or an address to a structure (such as &data). Bit 0 of the lowest byte in RAM is treated as the LSB.

**Availability:**
All devices

**Requires:**
Nothing

**Examples:**
```c
byte buffer[3];
for(i=0; i<=24; ++i){
    // Wait for clock high
    while (!input(PIN_A2));
    shift_left(buffer,3,input(PIN_A3));
    // Wait for clock low
    while (input(PIN_A2));
}
// reads 24 bits from pin A3, each bit is read
// on a low to high on pin A2
```

**Example Files:**
ex_extee.c, 9356.c

**Also See:**
shift_right(), rotate_right(), rotate_left().

---

# shift_right( )

**Syntax:**

```plaintext
shift_right (address, bytes, value)
```

**Parameters:**
- `address` is a pointer to memory.
- `bytes` is a count of the number of bytes to work with.
- `value` is a 0 to 1 to be shifted in.
## Built-in Functions

<table>
<thead>
<tr>
<th>Returns</th>
<th>0 or 1 for the bit shifted out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Shifts a bit into an array or structure. The address may be an array identifier or an address to a structure (such as &amp;data). Bit 0 of the lowest byte in RAM is treated as the LSB.</td>
</tr>
<tr>
<td>Availability</td>
<td>All devices</td>
</tr>
<tr>
<td>Requires</td>
<td>Nothing</td>
</tr>
</tbody>
</table>
| Examples | // reads 16 bits from pin A1, each bit is read  
// on a low to high on pin A2  
struct {  
byte time;  
byte command : 4;  
byte source : 4;} msg;

    for(i=0; i<=16; ++i) {
        while(!input(PIN_A2));  
        shift_right(&msg,3,input(PIN_A1));  
        while (input(PIN_A2)) ;}

    // This shifts 8 bits out PIN_A0, LSB first.  
    for(i=0;i<8;++i)  
        output_bit(PIN_A0,shift_right(&data,1,0)); |
| Example Files | ex_extee.c, 9356.c |
| Also See | shift_left(), rotate_right(), rotate_left(), sleep( ) |

### sleep( )

**Syntax:**  sleep(mode)

**Parameters:**  
- `mode` configures what sleep mode to enter, mode is optional. If mode is SLEEP_IDLE, the PIC will stop executing code but the peripherals will still be operational. If mode is SLEEP_FULL, the PIC will stop executing code and the peripherals will stop being clocked, peripherals that do not need a clock or are using an external clock will still be operational. SLEEP_FULL will reduce power consumption the most. If no parameter is specified, SLEEP_FULL will be used.
Returns: Undefined

Function: Issues a SLEEP instruction. Details are device dependent. However, in general the part will enter low power mode and halt program execution until woken by specific external events. Depending on the cause of the wake up execution may continue after the sleep instruction. The compiler inserts a sleep() after the last statement in main().

Availability: All devices

Requires: Nothing

Examples: disable_interruptions(INT_GLOBAL);
enable_interrupt(INT_EXT);
clear_interrupt();
sleep(SLEEP_FULL); //sleep until an INT_EXT interrupt
//after INT_EXT wake-up, will resume operation from this point

Example Files: ex_wakup.c

Also See: reset cpu()

sleep_ulpwu( )

Syntax: sleep_ulpwu(time)

Parameters: time specifies how long, in us, to charge the capacitor on the ultra-low power wakeup pin (by outputting a high on PIN_B0).

Returns: undefined

Function: Charges the ultra-low power wake-up capacitor on PIN_B0 for time microseconds, and then puts the PIC to sleep. The PIC will then wake-up on an 'Interrupt-on-Change' after the charge on the cap is lost.

Availability: Ultra Low Power Wake-Up support on the PIC (example, PIC124F32KA302)

Requires: #USE DELAY
### Examples

```c
while(TRUE)
{
    if (input(PIN_A1))
        //do something
    else
        sleepUlpmu(10);  //cap will be charged for 10us, //then goto sleep
}
```

<table>
<thead>
<tr>
<th>Example Files:</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Also See:</td>
<td>#USE_DELAY</td>
</tr>
</tbody>
</table>

---

## smtx_read()

**Syntax:**

```c
value_smt1_read(which);
value_smt2_read(which);
```

**Parameters:**

- `which`: Specifies which SMT registers to read. The following defines have been made in the device's header file to select which registers are read:
  - SMT_CAPTURED_PERIOD_REG
  - SMT_CAPTURED_PULSE_WIDTH_REG
  - SMT_TMR_REG
  - SMT_PERIOD_REG

**Returns:**

32-bit value

**Function:** To read the Capture Period Registers, Capture Pulse Width Registers, Timer Registers or Period Registers of the Signal Measurement Timer module.

**Availability:** Only devices with a built-in SMT module.

**Examples:**

```c
unsigned int32 Period;
Period = smt1_read(SMT_CAPTURED_PERIOD_REG);
```

<table>
<thead>
<tr>
<th>Example Files:</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Also See:</td>
<td>smtx_status(), stmx_start(), smtx_stop(), smtx_update(), smtx_reset_timer(), setup_SMTx(), smtx_write()</td>
</tr>
</tbody>
</table>
## smtx_reset_timer()

**Syntax:**

```c
smt1_reset_timer();
smt2_reset_timer();
```

**Parameters:**

None

**Returns:**

Nothing

**Function:**

To manually reset the Timer Register of the Signal Measurement Timer module.

**Availability:**

Only devices with a built-in SMT module.

**Examples:**

```
smt1_reset_timer();
```

**Example Files:**

None

**Also See:**

`setup_smtx()`, `smtx_start()`, `smtx_stop()`, `smtx_update()`, `smtx_status()`, `smtx_read()`, `smtx_write()`

## smtx_start()

**Syntax:**

```c
smt1_start();
smt2_start();
```

**Parameters:**

None

**Returns:**

Nothing

**Function:**

To have the Signal Measurement Timer (SMT) module start acquiring data.

**Availability:**

Only devices with a built-in SMT module.

**Examples:**

```
smt1_start();
```

**Example Files:**

None

**Also See:**

`smtx_status()`, `setup_smtx()`, `smtx_stop()`, `smtx_update()`, `smtx_reset_timer()`
**smtx_status( )**

Syntax:  
```
value = smt1_status();
value = smt2_status();
```

Parameters:  
None

Returns:  
The status of the SMT module.

Function:  
To return the status of the Signal Measurement Timer (SMT) module.

Availability:  
Only devices with a built-in SMT module.

Examples:  
```
status = smt1_status();
```

Also See:  
```
ssetup_smtx(), smtx_start(), smtx_stop(), smtx_update(),
smtx_reset_timer(),
smtx_read(), smtx_write()
```

**smtx_stop( )**

Syntax:  
```
smt1_stop();
smt2_stop();
```

Parameters:  
None

Returns:  
Nothing

Function:  
Configures the Signal Measurement Timer (SMT) module.

Availability:  
Only devices with a built-in SMT module.

Examples:  
```
smt1_stop()
```

Example Files:  
None
### smtx_write( )

**Syntax:**
```
smt1_write(which,value);
smt2_write(which,value);
```

**Parameters:**
- **which** - Specifies which SMT registers to write. The following defines have been made in the device's header file to select which registers are written:
  - SMT_TMR_REG
  - SMT_PERIOD_REG

- **value** - The 24-bit value to set the specified registers.

**Returns:**
Nothing

**Function:**
To write the Timer Registers or Period Registers of the Signal Measurement Timer (SMT) module

**Availability:**
Only devices with a built-in SMT module.

**Examples:**
```
smt1_write(SMT_PERIOD_REG, 0x10000000);
```

**Example Files:**
None

**Also See:**
smtx_status(), stmx_start(), setup_smtx(), smtx_update(), smtx_reset_timer(), smtx_read(), setup_smtx()
**smtx_update( )**

**Syntax:**
```c
smt1_update(which);
smt2_update(which);
```

**Parameters:**
- `which` - Specifies which capture registers to manually update. The following defines have been made in the device's header file to select which registers are updated:
  ```
  SMT_CAPTURED_PERIOD_REG
  SMT_CAPTURED_PULSE_WIDTH_REG
  ```

**Returns:**
- Nothing

**Function:**
To manually update the Capture Period Registers or the Capture Pulse Width Registers of the Signal Measurement Timer module.

**Availability:**
- Only devices with a built-in SMT module.

**Examples:**
```c
smt1_update(SMT_CAPTURED_PERIOD_REG);
```

**Example Files:**
- None

**Also See:**
- `setup_smtx()`, `stmx_start()`, `smtx_stop()`, `smtx_status()`, `smtx_reset_timer()`, `smtx_read()`, `smtx_write()`

---

**spi_data_is_in( )**

**Syntax:**
```c
result = spi_data_is_in()
result = spi_data_is_in2()
```

**Parameters:**
- None

**Returns:**
- 0 (FALSE) or 1 (TRUE)

**Function:**
Returns TRUE if data has been received over the SPI.

**Availability:**
- This function is only available on devices with SPI hardware.

**Requires:**
- Nothing
Examples:

( !spi_data_is_in() && input(PIN_B2) );
if( spi_data_is_in() )
data = spi_read();

Example Files:
None
Also See: spi_read(), spi_write(), SPI Overview

---

**spi_init()**

Syntax:

spi_init(baud);
spi_init(stream, baud);

Parameters:

- **stream** – is the SPI stream to use as defined in the STREAM=name option in #USE SPI.
- **band** – the band rate to initialize the SPI module to. If FALSE it will disable the SPI module, if TRUE it will enable the SPI module to the band rate specified in #use SPI.

Returns: Nothing.

Function: Initializes the SPI module to the settings specified in #USE SPI.

Availability: This function is only available on devices with SPI hardware.

Requires: #USE SPI

Examples:

#use spi(MATER, SPI1, baud=1000000, mode=0, stream=SPI1_MODE0)

spi_init(SPI1_MODE0, TRUE); //initialize and enable SPI1 to setting in #USE SPI
spi_init(FALSE); //disable SPI1
spi_init(250000); //initialize and enable SPI1 to a baud rate of 250K

Example Files:
None
Also See: #USE SPI, spi_xfer(), spi_xfer_in(), spi_prewrite(), spi_speed()
### spi_prewrite(data);

| Syntax: | spi_prewrite(data);
|         | spi_prewrite(stream, data); |
| Parameters: | stream – is the SPI stream to use as defined in the STREAM=name option in #USE SPI. |
|           | data - the variable or constant to transfer via SPI |
| Returns: | Nothing. |
| Function: | Writes data into the SPI buffer without waiting for transfer to be completed. Can be used in conjunction with spi_xfer() with no parameters to transfer more then 8 bits for PCM and PCH device, or more then 8 bits or 16 bits (XFER16 option) for PCD. Function is useful when using the SSP or SSP2 interrupt service routines for PCM and PCH device, or the SPIx interrupt service routines for PCD device. |
| Availability: | This function is only available on devices with SPI hardware. |
| Requires: | #USE SPI, and the option SLAVE is used in #USE SPI to setup PIC as a SPI slave device |
| Examples: | spi_prewrite(data_out); |
| Example Files: | ex_spi_slave.c |
| Also See: | #USE SPI, spi_xfer(), spi_xfer_in(), spi_init(), spi_speed() |

### spi_read( )  spi_read2()
### spi_read3( )  spi_read4()

| Syntax: | value = spi_read ([data])
|         | value = spi_read2 ([data])
|         | value = spi_read3([data])
|         | value = spi_read4 ([data]) |
| Parameters: | data – optional parameter and if included is an 8 bit int. |
| Returns: | An 8 bit int |
| Function: | Return a value read by the SPI. If a value is passed to the spi_read()
the data will be clocked out and the data received will be returned. If no data is ready, spi_read() will wait for the data is a SLAVE or return the last DATA clocked in from spi_write().

If this device is the MASTER then either do a spi_write(data) followed by a spi_read() or do a spi_read(data). These both do the same thing and will generate a clock. If there is no data to send just do a spi_read(0) to get the clock.

If this device is a SLAVE then either call spi_read() to wait for the clock and data or use_spi_data_is_in() to determine if data is ready.

Availability: This function is only available on devices with SPI hardware.

Requires: Nothing

Examples: data_in = spi_read(out_data);

Example Files: ex_spi.c

Also See: spi_write(), spi_write_16(), spi_read_16(), spi_data_is_in(), SPI Overview

---

**spi_read_16()**  **spi_read2_16()**  **spi_read3_16()**  **spi_read4_16()**

Syntax: value = spi_read_16([data]);
value = spi_read2_16([data]);
value = spi_read3_16([data]);
value = spi_read4_16([data]);

Parameters: data – optional parameter and if included is a 16 bit int

Returns: A 16 bit int

Function: Return a value read by the SPI. If a value is passed to the spi_read_16() the data will be clocked out and the data received will be returned. If no data is ready, spi_read_16() will wait for the data is a SLAVE or return the last DATA clocked in from spi_write_16(). If this device is the MASTER then either do a spi_write_16(data) followed
by a `spi_read_16()` or do a `spi_read_16(data)`. These both do the same thing and will generate a clock. If there is no data to send just do a `spi_read_16(0)` to get the clock. If this device is a slave then either call `spi_read_16()` to wait for the clock and data or `use_spi_data_is_in()` to determine if data is ready.

**Availability:** This function is only available on devices with SPI hardware.

**Requires:** That the option `SPI_MODE_16B` be used in `setup_spi()` function, or that the option `XFER16` be used in `#use SPI`.

**Examples:**
```
data_in = spi_read_16(out_data);
```

**Example Files:** None

**Also See:** `spi_read()`, `spi_write()`, `spi_write_16()`, `spi_data_is_in()`, SPI Overview

---

**spi_speed**

**Syntax:**
```
spi_speed(baud);
spi_speed(stream, baud);
spi_speed(stream, baud, clock);
```

**Parameters:**
- `stream` – is the SPI stream to use as defined in the `STREAM=name` option in `#USE SPI`.
- `band` - the band rate to set the SPI module to.
- `clock` - the current clock rate to calculate the band rate with.

If not specified it uses the value specified in `#use delay()`.

**Returns:** Nothing.

**Function:** Sets the SPI module's baud rate to the specified value.

**Availability:** This function is only available on devices with SPI hardware.

**Requires:** `#USE SPI`

**Examples:**
```
spi_speed(250000);
spi_speed(SPI1_MODE0, 250000);
spi_speed(SPI1_MODE0, 125000, 8000000);
```

**Example Files:** None

**Also See:** `#USE SPI`, `spi_xfer()`, `spi_xfer_in()`, `spi_prewrite()`, `spi_init()`
### spi_write( )  spi_write2( )  spi_write3( )  spi_write4( )

**Syntax:**

- `spi_write([wait], value);`
- `spi_write2([wait], value);`
- `spi_write3([wait], value);`
- `spi_write4([wait], value);`

**Parameters:**

- `value` is an 8 bit int
- `wait` - an optional parameter specifying whether the function will wait for the SPI transfer to complete before exiting. Default is TRUE if not specified.

**Returns:**

Nothing

**Function:**

Sends a byte out the SPI interface. This will cause 8 clocks to be generated. This function will write the value out to the SPI. At the same time data is clocked out data is clocked in and stored in a receive buffer. `spi_read()` may be used to read the buffer.

**Availability:**

This function is only available on devices with SPI hardware.

**Requires:**

Nothing

**Examples:**

```c
spi_write( data_out );
data_in = spi_read();
```

**Example Files:**

- `ex_spi.c`

**Also See:**

- `spi_read()`, `spi_data_is_in()`, `SPI Overview`, `spi_write_16()`, `spi_read_16()`

---

### spi_xfer( )

**Syntax:**

- `spi_xfer(data)`
- `spi_xfer(stream, data)`
- `spi_xfer(stream, data, bits)`
- `result = spi_xfer(data)`
- `result = spi_xfer(stream, data)`
- `result = spi_xfer(stream, data, bits)`
### Parameters:

- **data** is the variable or constant to transfer via SPI. The pin used to transfer data is defined in the DO=pin option in `#use spi`.

- **stream** is the SPI stream to use as defined in the STREAM=name option in `#USE SPI`.

- **bits** is how many bits of data will be transferred.

### Returns:

The data read in from the SPI. The pin used to transfer result is defined in the DI=pin option in `#USE SPI`.

### Function:

Transfers data to and reads data from an SPI device.

### Availability:

All devices with SPI support.

### Requires:

`#USE SPI`

### Examples:

```c
int i = 34;
spi_xfer(i);
// transfers the number 34 via SPI
int trans = 34, res;
res = spi_xfer(trans);
// transfers the number 34 via SPI
// also reads the number coming in from SPI
```

### Also See:

`#USE_SPI`

---

### SPI_XFER_IN()

#### Syntax:

```c
value = spi_xfer_in();
value = spi_xfer_in(bits);
value = spi_xfer_in(stream, bits);
```

#### Parameters:

- **stream** – is the SPI stream to use as defined in the STREAM=name option in `#USE SPI`.
- **bits** – is how many bits of data to be received.

#### Returns:

The data read in from the SPI

#### Function:

Reads data from the SPI, without writing data into the transmit buffer
### `sprintf()`

**Syntax:**
```
sprintf(string, cstring, values...);
bytes=sprintf(string, cstring, values...)
```

**Parameters:**
- `string` is an array of characters.
- `cstring` is a constant string or an array of characters null terminated.
- `Values` are a list of variables separated by commas. Note that format specifies do not work in ram band strings.

**Returns:**
Bytes is the number of bytes written to string.

**Function:**
This function operates like printf() except that the output is placed into the specified string. The output string will be terminated with a null. No checking is done to ensure the string is large enough for the data. See printf() for details on formatting.

**Availability:**
All devices.

**Requires:**
Nothing

**Examples:**
```
char mystring[20];
long mylong;

mylong=1234;
sprintf(mystring,"<%lu">",mylong);
// mystring now has:
// < 1 2 3 4 > \0
```

**Example**
None
sqrt( )

Syntax: \( \text{result} = \text{sqrt}(\text{value}) \)

Parameters: \( \text{value} \) is any float type

Returns: Returns a floating point value with a precision equal to \( \text{value} \)

Function: Computes the non-negative square root of the float value \( x \). If the argument is negative, the behavior is undefined.

Note on error handling:
If "errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Domain error occurs in the following cases:
sqrt: when the argument is negative

Availability: All devices.

Requires: #INCLUDE <math.h>

Examples: \( \text{distance} = \text{sqrt}( \text{pow}((\text{x1}-\text{x2}),2)+\text{pow}((\text{y1}-\text{y2}),2) ) ; \)

Example Files: None

Also See: None

srand( )

Syntax: srand(\( n \))

Parameters: \( n \) is the seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand.
Returns: No value.

Function: The srand() function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand. If srand() is then called with same seed value, the sequence of random numbers shall be repeated. If rand is called before any call to srand() have been made, the same sequence shall be generated as when srand() is first called with a seed value of 1.

Availability: All devices.

Requires: #INCLUDE <STDLIB.H>

Examples:
```c
srand(10);
I=rand();
```

Example Files: None

Also See: rand()

---

**STANDARD STRING FUNCTIONS( )**

- memchr( )
- memcmp( )
- strcat( )
- strchr( )
- strcmp( )
- strcoll( )
- strcspn( )
- strncmp( )
- strncpy( )
- strpbrk( )
- strrchr( )
- strspn( )
- strstr( )
- strxfrm( )

**Syntax:**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr=strcat (s1, s2)</td>
<td>Concatenate s2 onto s1</td>
</tr>
<tr>
<td>ptr=strchr (s1, c)</td>
<td>Find c in s1 and return &amp;s1[i]</td>
</tr>
<tr>
<td>ptr=strchr (s1, c)</td>
<td>Same but search in reverse</td>
</tr>
<tr>
<td>result=strcmp (s1, s2)</td>
<td>Compare s1 to s2</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>iresult=strncmp (s1, s2, n)</code></td>
<td>Compare s1 to s2 (n bytes)</td>
</tr>
<tr>
<td><code>iresult=stricmp (s1, s2)</code></td>
<td>Compare and ignore case</td>
</tr>
<tr>
<td><code>ptr=strncpy (s1, s2, n)</code></td>
<td>Copy up to n characters s2-&gt;s1</td>
</tr>
<tr>
<td><code>iresult=strcsn (s1, s2)</code></td>
<td>Count of initial chars in s1 not in s2</td>
</tr>
<tr>
<td><code>iresult=strspn (s1, s2)</code></td>
<td>Count of initial chars in s1 also in s2</td>
</tr>
<tr>
<td><code>iresult=strlen (s1)</code></td>
<td>Number of characters in s1</td>
</tr>
<tr>
<td><code>ptr=strlwr (s1)</code></td>
<td>Convert string to lower case</td>
</tr>
<tr>
<td><code>ptr=strpbrk (s1, s2)</code></td>
<td>Search s1 for first char also in s2</td>
</tr>
<tr>
<td><code>ptr=strstr (s1, s2)</code></td>
<td>Search for s2 in s1</td>
</tr>
<tr>
<td><code>ptr=strncat (s1, s2, n)</code></td>
<td>Concatenates up to n bytes of s2 onto s1</td>
</tr>
<tr>
<td><code>iresult=strcoll (s1, s2)</code></td>
<td>Compares s1 to s2, both interpreted as appropriate to the current locale.</td>
</tr>
<tr>
<td><code>res=strxfrm (s1, s2, n)</code></td>
<td>Transforms maximum of n characters of s2 and places them in s1, such that strcpy(s1, s2) will give the same result as strcoll(s1, s2)</td>
</tr>
<tr>
<td><code>iresult=memcmp (m1, m2, n)</code></td>
<td>Compare m1 to m2 (n bytes)</td>
</tr>
<tr>
<td><code>ptr=memchr (m1, c, n)</code></td>
<td>Find c in first n characters of m1 and return &amp;m1[i]</td>
</tr>
<tr>
<td><code>ptr=strerror (errnum)</code></td>
<td>Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.</td>
</tr>
</tbody>
</table>

**Parameters:**

- `s1` and `s2` are pointers to an array of characters (or the name of an array). Note that `s1` and `s2` MAY NOT BE A CONSTANT (like "hi").
- `n` is a count of the maximum number of character to operate on.
- `c` is a 8 bit character.
- `m1` and `m2` are pointers to memory.

**Returns:**

- `ptr` is a copy of the `s1` pointer
- `iresult` is an 8 bit int
- result is -1 (less than), 0 (equal) or 1 (greater than)
- `res` is an integer.

**Function:**

Functions are identified above.

**Availability:**

All devices.

**Requires:**

`#include <string.h>`
## Examples:
```c
char string1[10], string2[10];
strcpy(string1, "hi ");
strcpy(string2, "there");
strcat(string1, string2);  
printf("Length is %u\r\n", strlen(string1));
// Will print 8
```

### Files:
- `ex_str.c`  
### Also See:
- `strcpy()`, `strtok()`  

---

### `strcpy( )`  
### `strcopy( )`

#### Syntax:
- `strcpy (dest, src)`
- `strcopy (dest, src)`

#### Parameters:
- `dest` is a pointer to a RAM array of characters.
- `src` may be either a pointer to a RAM array of characters or it may be a constant string.

#### Returns:
undefined  

#### Function:
Copies a constant or RAM string to a RAM string. Strings are terminated with a 0.

#### Availability:
All devices.

#### Requires:
Nothing  

#### Examples:
```c
char string[10], string2[10];
.
.
strcpy (string, "Hi There");
strcpy(string2,string);
```

### Files:
- `ex_str.c`  
### Also See:
- `strxxxx()`
## strtod( ) strtof( ) strtof48( )

### Syntax:
- `result=strtod(nptr,& endptr)`
- `result=strtof(nptr,& endptr)`
- `result=strtof48(nptr,& endptr)`

### Parameters:
- `nptr` and `endptr` are strings

### Returns:
- `strtod` returns a double precision floating point number.
- `strtof` returns a single precision floating point number.
- `strtof48` returns an extended precision floating point number.

### Function:
The `strtod` function converts the initial portion of the string pointed to by `nptr` to a float representation. The part of the string after conversion is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer. If `nptr` is empty or does not have the expected form, no conversion is performed and the value of `nptr` is stored in the object pointed to by `endptr`, provided `endptr` is not a null pointer.

### Availability:
All devices.

### Requires:
`# INCLUDE <stdlib.h>`

### Examples:
```c
double result;
char str[12]="123.45hello";
char *ptr;
result=strtod(str,&ptr);
//result is 123.45 and ptr is "hello"
```

### Example Files:
None

### Also See:
`strtol()`, `strtoul()`

## strtok( )

### Syntax:
`ptr = strtok(s1, s2)`

### Parameters:
- `s1` and `s2` are pointers to an array of characters (or the name of an array). Note that `s1` and `s2` MAY NOT BE A CONSTANT (like "hi"). `s1`
may be 0 to indicate a continue operation.

**Returns:** ptr points to a character in s1 or is 0

**Function:** Finds next token in s1 delimited by a character from separator string s2 (which can be different from call to call), and returns pointer to it.

First call starts at beginning of s1 searching for the first character NOT contained in s2 and returns null if there is none are found.

If none are found, it is the start of first token (return value). Function then searches from there for a character contained in s2.

If none are found, current token extends to the end of s1, and subsequent searches for a token will return null.

If one is found, it is overwritten by '\0', which terminates current token. Function saves pointer to following character from which next search will start.

Each subsequent call, with 0 as first argument, starts searching from the saved pointer.

**Availability:** All devices.

**Requires:** #INCLUDE <string.h>

**Examples:**

```c
char string[30], term[3], *ptr;
strcpy(string,"one,two,three; ");
strcpy(term,"; ");

ptr = strtok(string, term);
while(ptr!=0) {
    puts(ptr);
    ptr = strtok(0, term);
}

// Prints:
one
two
three
```

**Example Files:** ex_str.c

**Also See:** strxxx(), strcpy()
## strtol( )

**Syntax:**

\[
\text{result} = \text{strtol}(nptr, & \text{endptr}, \text{base})
\]

**Parameters:**

- `nptr` and `endptr` are strings and `base` is an integer

**Returns:**

- `result` is a signed long int.
- Returns the converted value in `result`, if any. If no conversion could be performed, zero is returned.

**Function:**

The `strtol` function converts the initial portion of the string pointed to by `nptr` to a signed long int representation in some radix determined by the value of `base`. The part of the string after conversion is stored in the object pointed to `endptr`, provided that `endptr` is not a null pointer. If `nptr` is empty or does not have the expected form, no conversion is performed and the value of `nptr` is stored in the object pointed to by `endptr`, provided `endptr` is not a null pointer.

**Availability:**

All devices.

**Requires:**

`#include <stdlib.h>`

**Examples:**

```c
signed long result;
char str[9]="123hello";
char *ptr;
result=strtol(str,&ptr,10);
//result is 123 and ptr is "hello"
```

**Example Files:**

None

**Also See:**

`strtod()`, `strtol()`

## strtoul( )

**Syntax:**

\[
\text{result} = \text{strtoul}(nptr, \text{endptr}, \text{base})
\]

**Parameters:**

- `nptr` and `endptr` are strings pointers and `base` is an integer 2-36.
Returns: result is an unsigned long int. returns the converted value in result , if any. If no conversion could be performed, zero is returned.

Function: The strtoul function converts the initial portion of the string pointed to by nptr to a long int representation in some radix determined by the value of base. The part of the string after conversion is stored in the object pointed to endptr, provided that endptr is not a null pointer. If nptr is empty or does not have the expected form, no conversion is performed and the value of nptr is stored in the object pointed to by endptr, provided endptr is not a null pointer.

Availability: All devices.

Requires: STDLIB.H must be included

Examples: long result;
char str[9]="123hello";
char *ptr;
result=strtoul(str,&ptr,10);
//result is 123 and ptr is "hello"

Example Files: None

Also See: strtol(), strtod()

---

**swap( )**

Syntax: swap (lvalue)
result = swap(lvalue)

Parameters: lvalue is a byte variable

Returns: A byte

Function: Swaps the upper nibble with the lower nibble of the specified byte. This is the same as:
byte = (byte << 4) | (byte >> 4);
# Built-in Functions

<table>
<thead>
<tr>
<th>Availability:</th>
<th>All devices.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires:</td>
<td>Nothing</td>
</tr>
<tr>
<td>Examples:</td>
<td></td>
</tr>
</tbody>
</table>

```c
x = 0x45;
swap(x);
// x now is 0x54

int x = 0x42;
int result;
result = swap(x);
// result is 0x24
```

<table>
<thead>
<tr>
<th>Example Files:</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Also See:</td>
<td><code>rotate_right()</code>, <code>rotate_left()</code></td>
</tr>
</tbody>
</table>

## tolower( )

### Syntax:

```c
result = tolower (cvalue)  
result = toupper (cvalue)
```

### Parameters:

- `cvalue` is a character

### Returns:

An 8 bit character

### Function:

These functions change the case of letters in the alphabet.

TOLOWER(X) will return 'a'..'z' for X in 'A'..'Z' and all other characters are unchanged. TOUPPER(X) will return 'A'..'Z' for X in 'a'..'z' and all other characters are unchanged.

### Availability:

All devices.

### Requires:

Nothing

### Examples:

```c
switch( toupper(getc()) ) {
    case 'R' : read_cmd();   break;
    case 'W' : write_cmd();  break;
    case 'Q' : done=TRUE;    break;
    ...
}
```
### `touchpad_getc()`

**Syntax:**

```c
input = TOUCHPAD_GETC();
```

**Parameters:**

None

**Returns:**

char (returns corresponding ASCII number is “input” declared as int)

**Function:**

Actively waits for firmware to signal that a pre-declared Capacitive Sensing Module (CSM) or charge time measurement unit (CTMU) pin is active, then stores the pre-declared character value of that pin in “input”.

Note: Until a CSM or CTMU pin is read by firmware as active, this instruction will cause the microcontroller to stall.

**Availability:**

All PIC’s with a CSM or CTMU Module

**Requires:**

```c
#define TOUCHPAD (options)
```

**Examples:**

```c
//When the pad connected to PIN_B0 is activated, store the letter 'A'

#define TOUCHPAD (PIN_B0='A')
void main(void){
    char c;
    enable_interrupts(GLOBAL);

    c = TOUCHPAD_GETC();
    //will wait until one of declared pins is detected
    //if PIN_B0 is pressed, c will get value 'A'
}
```

**Example Files:**

None
### touchpad_hit( )

**Syntax:**  
value = TOUCHPAD_HIT()

**Parameters:** None

**Returns:** TRUE or FALSE

**Function:** Returns TRUE if a Capacitive Sensing Module (CSM) or Charge Time Measurement Unit (CTMU) key has been pressed. If TRUE, then a call to touchpad_getc() will not cause the program to wait for a key press.

**Availability:** All PIC’s with a CSM or CTMU Module

**Requires:** #USE TOUCHPAD (options)

**Examples:**

```c
// When the pad connected to PIN_B0 is activated, store the letter 'A'

#USE TOUCHPAD (PIN_B0='A')
void main(void){
    char c;
    enable_interruption(GLOBAL);

    while (TRUE) {
        if ( TOUCHPAD_HIT() )
            // wait until key on PIN_B0 is pressed
            c = TOUCHPAD_GETC();  // get key that was pressed
            // c will get value 'A'
    }
}
```

**Example Files:** None

**Also See:** #USE TOUCHPAD, touchpad_state(), touchpad_getc()
Returns: None

Function: Sets the current state of the touchpad connected to the Capacitive Sensing Module (CSM). The state can be one of the following three values:

0: Normal state
1: Calibrates, then enters normal state
2: Test mode, data from each key is collected in the int16 array TOUCHDATA

Note: If the state is set to 1 while a key is being pressed, the touchpad will not calibrate properly.

Availability: All PIC's with a CSM Module

Requires: #USE TOUCHPAD (options)

Examples:
```
#USE TOUCHPAD (THRESHOLD=5, PIN_D5='5', PIN_B0='C')
void main(void){
    char c;
    TOUCHPAD_STATE(1); //calibrates, then enters normal state
    enable_interrupts(GLOBAL);
    while(1){
        c = TOUCHPAD_GETC();
        //will wait until one of declared pins is detected
    }
    //if PIN_B0 is pressed, c will get value 'C'
}
```

Example Files: None

Also See: #USE TOUCHPAD, touchpad_getc(), touchpad_hit()
### tx_buffer_available()

**Syntax:**

```plaintext
value = tx_buffer_available([stream]);
```

**Parameters:**

- `stream` – optional parameter specifying the stream defined in `#USE RS232`.

**Returns:**

Number of bytes that can still be put into transmit buffer

**Function:**

Function to determine the number of bytes that can still be put into transmit buffer before it overflows. Transmit buffer is implemented as a circular buffer, so be sure to check to make sure there is room for at least one more than what is actually needed.

**Availability:**

All devices

**Requires:**

`#USE RS232`

**Examples:**

```c
#USE_RS232(UART1, BAUD=9600, TRANSMIT_BUFFER=50)
void main(void) {
    unsigned int8 Count = 0;

    while(TRUE) {
        if(tx_buffer_available()>13)
            printf("/r/nCount=%3u", Count++);
    }
}
```

**Example Files:**

None

**Also See:**

`_USE_RS232()`, `rcv()`, `TX_BUFFER_FULL()`, `RCV_BUFFER_BYTES()`, `GET()`, `PUTC()`, `PRINTF()`, `SETUP_UART()`, `PUTC_SEND()`

### tx_buffer_bytes()

**Syntax:**

```plaintext
value = tx_buffer_bytes([stream]);
```

**Parameters:**

- `stream` – optional parameter specifying the stream defined in `#USE RS232`.

---

473
**Returns:** Number of bytes in transmit buffer that still need to be sent.

**Function:** Function to determine the number of bytes in transmit buffer that still need to be sent.

**Availability:** All devices

**Requires:** #USE RS232

**Examples:**

```c
USE_RS232(UART1,BAUD=9600,TRANSMITBUFFER=50)
void main(void) {
    char string[] = "Hello";
    if(tx_buffer_bytes() <= 45)
        printf("%s",string);
}
```

**Example Files:** None

**Also See:** USE_RS232(), RCV_BUFFER_FULL(), TX_BUFFER_FULL(), RCV_BUFFER_BYTES(), GET(), PUTC(), PRINTF(), SETUP_UART(), PUTC_SEND()

---

### tx_buffer_full()

**Syntax:** `value = tx_buffer_full([stream])`

**Parameters:**

- `stream` – optional parameter specifying the stream defined in #USE RS232

**Returns:** TRUE if transmit buffer is full, FALSE otherwise.

**Function:** Function to determine if there is room in transmit buffer for another character.

**Availability:** All devices

**Requires:** #USE RS232
Examples:  

```c
#USE_RS232(UART1,BAUD=9600,TRANSMIT_BUFFER=50)
void main(void) {
    char c;

    if(!tx_buffer_full())
        putc(c);
}
```

Example Files: None

Also See:  

_USE_RS232( ), RCV_BUFFER_FULL( ), TX_BUFFER_FULL( ), RCV_BUFFER_BYTES( ), GETC( ), PUTC( ), PRINTF( ), SETUP_UART( ), PUTC_SEND( )

---

**va_arg( )**

**Syntax:**

```c
va_arg(argptr, type)
```

**Parameters:**

- `argptr` is a special argument pointer of type `va_list`
- `type` – This is data type like int or char.

**Returns:**

The first call to `va_arg` after `va_start` return the value of the parameters after that specified by the last parameter. Successive invocations return the values of the remaining arguments in succession.

**Function:**

The function will return the next argument every time it is called.

**Availability:**

All devices.

**Requires:**

```c
#include <stdarg.h>
```

**Examples:**

```c
int foo(int num, ...)
{
    int sum = 0;
    int i;
    va_list argptr; // create special argument pointer
    va_start(argptr, num); // initialize argptr
    for(i=0; i<num; i++)
        sum = sum + va_arg(argptr, int);
```
va_end(argptr);  // end variable processing
return sum;
}

### va_end()

**Syntax:**
va_end(argptr)

**Parameters:**
argptr is a special argument pointer of type va_list.

**Returns:**
None

**Function:**
A call to the macro will end variable processing. This will facilitate a normal return from the function whose variable argument list was referred to by the expansion of va_start().

**Availability:**
All devices.

**Requires:**
#INCLUDE <stdarg.h>

**Examples:**
```c
int foo(int num, ...)
{
    int sum = 0;
    int i;
    va_list argptr;  // create special argument pointer
    va_start(argptr,num);  // initialize argptr
    for(i=0; i<num; i++)
        sum = sum + va_arg(argptr, int);
    va_end(argptr);  // end variable processing
    return sum;
}
```

**Example Files:**
None

**Also See:**
nargs(), va_end(), va_start()
### va_start

**Syntax:**

```
va_start(argptr, variable)
```

**Parameters:**

- `argptr` is a special argument pointer of type `va_list`

- `variable` – The second parameter to `va_start()` is the name of the last parameter before the variable-argument list.

**Returns:**

None

**Function:**

The function will initialize the `argptr` using a call to the macro `va_start()`.

**Availability:**

All devices.

**Requires:**

```
#include <stdarg.h>
```

**Examples:**

```c
int foo(int num, ...)
{
    int sum = 0;
    int i;
    va_list argptr; // create special argument pointer
    va_start(argptr,num); // initialize argptr
    for(i=0; i<num; i++)
        sum = sum + va_arg(argptr, int);
    va_end(argptr);    // end variable processing
    return sum;
}
```

**Example Files:**

None

**Also See:**

`nargs()`, `va_start()`, `va_arg()`

---

### write_configuration_memory( )

**Syntax:**

```
write_configuration_memory ([offset], dataptr,count)
```

**Parameters:**

- `dataptr`: pointer to one or more bytes
- `count`: a 8 bit integer
- `offset` is an optional parameter specifying the offset into configuration memory to start writing to, offset defaults to zero if not used.
**Returns:** undefined

**Function:** Erases all fuses and writes count bytes from the dataptr to the configuration memory.

**Availability:** All PIC24 Flash devices

**Requires:** Nothing

**Examples:**

```c
int data[6];
write_configuration_memory(data,6)
```

**Example Files:** None

**Also See:** write_program_memory(), Configuration Memory Overview

---

**write_eeprom( )**

**Syntax:**

```c
write_eeprom (address, value)
write_eeprom ( address , pointer , N )
```

**Parameters:**

- **address** is the 0 based starting location of the EEPROM write
- **N** specifies the number of EEPROM bytes to write
- **value** is a constant or variable to write to EEPROM
- **pointer** is a pointer to location to data to be written to EEPROM

**Returns:** undefined

**Function:** This function will write the specified value to the given address of EEPROM. If pointers are used than the function will write n bytes of data from the pointer to EEPROM starting at the value of address.

In order to allow interrupts to occur while using the write operation, use the #DEVICE option WRITE_EEPROM = NOINT. This will allow interrupts to occur while the write_eeprom() operations is polling the done bit to check if the write operations has completed. Can be used as long as no EEPROM operations are performed during an ISR.
Built-in Functions

| Availability: | This function is only available on devices with supporting hardware on chip. |
| Requires: | Nothing |
| Examples: | #define LAST_VOLUME 10 // Location in EEPROM
volume++;
write_eeprom(LAST_VOLUME, volume); |
| Example Files: | None |
| Also See: | read_eeprom(), write_program_eeprom(), read_program_eeprom(), data Eeprom Overview |

write_extended_ram( )

| Syntax: | write_extended_ram (page,address,data,count); |
| Parameters: | page – the page in extended RAM to write to address – the address on the selected page to start writing to data – pointer to the data to be written count – the number of bytes to write (0-32768) |
| Returns: | undefined |
| Function: | To write data to the extended RAM of the PIC. |
| Availability: | On devices with more then 30K of RAM. |
| Requires: | Nothing |
| Examples: | unsigned int8 data[8] =
{0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
write_extended_ram(1,0x0000,data,8); |
| Example Files: | None |
| Also See: | read_extended_ram(), Extended RAM Overview |
write_program_memory()

Syntax:  write_program_memory( address, dataptr, count );

Parameters:  
- **address** is 32 bits.
- **dataptr** is a pointer to one or more bytes
- **count** is a 16 bit integer on PIC16 and 16-bit for PIC18

Returns:  undefined

Function:  Writes count bytes to program memory from dataptr to address. This function is most effective when count is a multiple of FLASH_WRITE_SIZE, but count needs to be a multiple of four. Whenever this function is about to write to a location that is a multiple of FLASH_ERASE_SIZE then an erase is performed on the whole block. Due to the 24 bit instruction length on PCD parts, every fourth byte of data is ignored. Fill the ignored bytes with 0x00.

See Program EEPROM Overview for more information on program memory access

Availability:  Only devices that allow writes to program memory.

Requires:  Nothing

Examples:  
```c
for(i=0x1000;i<=0x1fff;i++) {
    value=read_adc();
    write_program_memory(i, value, 2);
    delay_ms(1000);
}
```

```c
int8 write_data[4] = {0x10,0x20,0x30,0x00};
write_program_memory (0x2000, write_data, 4);
```

Example Files:  None

---

zdc_status()

Syntax:  value=zcd_status()

Parameters:  **None**

Returns:  value - the status of the ZCD module. The following defines are
### Built-in Functions

Made in the device's header file and are as follows:

- ZCD_IS_SINKING
- ZCD_IS_SOURCING

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function:</strong></td>
<td>To determine if the Zero-Cross Detection (ZCD) module is currently sinking or sourcing current. If the ZCD module is setup to have the output polarity inverted, the value return will be reversed.</td>
</tr>
<tr>
<td><strong>Availability:</strong></td>
<td>All devices with a ZCD module.</td>
</tr>
<tr>
<td><strong>Examples:</strong></td>
<td>value=zcd_status();</td>
</tr>
<tr>
<td><strong>Example Files:</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Also See:</strong></td>
<td>setup_zcd()</td>
</tr>
</tbody>
</table>
STANDARD C INCLUDE FILES

errno.h

<table>
<thead>
<tr>
<th>errno.h</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EDOM</td>
<td>Domain error value</td>
</tr>
<tr>
<td>ERANGE</td>
<td>Range error value</td>
</tr>
<tr>
<td>errno</td>
<td>error value</td>
</tr>
</tbody>
</table>

float.h

<table>
<thead>
<tr>
<th>float.h</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_RADIX:</td>
<td>Radix of the exponent representation</td>
</tr>
<tr>
<td>FLT_MANT_DIG:</td>
<td>Number of base digits in the floating point significant</td>
</tr>
<tr>
<td>FLT_DIG:</td>
<td>Number of decimal digits, q, such that any floating point number with p decimal digits can be rounded into a floating point number with p radix b digits and back again without change to the q decimal digits.</td>
</tr>
<tr>
<td>FLT_MIN_EXP:</td>
<td>Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number.</td>
</tr>
<tr>
<td>FLT_MIN_10_EXP:</td>
<td>Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.</td>
</tr>
<tr>
<td>FLT_MAX_EXP:</td>
<td>Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number.</td>
</tr>
<tr>
<td>FLT_MAX_10_EXP:</td>
<td>Maximum negative integer such that 10 raised to that power is in the range representable finite floating-point numbers.</td>
</tr>
<tr>
<td>FLT_MAX:</td>
<td>Maximum representable finite floating point number.</td>
</tr>
<tr>
<td>FLT_EPSILON:</td>
<td>The difference between 1 and the least value greater than 1 that is representable in the given floating point type.</td>
</tr>
<tr>
<td>FLT_MIN:</td>
<td>Minimum normalized positive floating point number</td>
</tr>
<tr>
<td>DBL_MANT_DIG:</td>
<td>Number of base digits in the double significant</td>
</tr>
<tr>
<td>DBL_DIG:</td>
<td>Number of decimal digits, q, such that any double number with p decimal digits can be rounded into a double number with p radix b digits and back again without change to the q decimal digits.</td>
</tr>
<tr>
<td>DBL_MIN_EXP:</td>
<td>Minimum negative integer such that FLT_RADIX raised to that power</td>
</tr>
</tbody>
</table>

483
minus 1 is a normalized double number.

DBL_MIN_10_EXP: Minimum negative integer such that 10 raised to that power is in the range of normalized double numbers.

DBL_MAX_EXP: Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite double number.

DBL_MAX_10_EXP: Maximum negative integer such that 10 raised to that power is in the range of representable finite double numbers.

DBL_MAX: Maximum representable finite floating point number.

DBL_EPSILON: The difference between 1 and the least value greater than 1 that is representable in the given floating point type.

DBL_MIN: Minimum normalized positive double number.

LDBL_MANT_DIG: Number of base digits in the floating point significant
LDBL_DIG: Number of decimal digits, q, such that any floating point number with p radix b digits and back again without change to the q decimal digits.

LDBL_MIN_EXP: Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number.

LDBL_MIN_10_EXP: Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.

LDBL_MAX_EXP: Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number.

LDBL_MAX_10_EXP: Maximum negative integer such that 10 raised to that power is in the range of representable finite floating-point numbers.

LDBL_MAX: Maximum representable finite floating point number.

LDBL_EPSILON: The difference between 1 and the least value greater than 1 that is representable in the given floating point type.

LDBL_MIN: Minimum normalized positive floating point number.

limits.h

<table>
<thead>
<tr>
<th>limits.h</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_BIT:</td>
<td>Number of bits for the smallest object that is not a bit_field.</td>
</tr>
<tr>
<td>SCHAR_MIN:</td>
<td>Minimum value for an object of type signed char</td>
</tr>
<tr>
<td>SCHAR_MAX:</td>
<td>Maximum value for an object of type signed char</td>
</tr>
<tr>
<td>UCHAR_MAX:</td>
<td>Maximum value for an object of type unsigned char</td>
</tr>
<tr>
<td>CHAR_MIN:</td>
<td>Minimum value for an object of type char(unsigned)</td>
</tr>
<tr>
<td>CHAR_MAX:</td>
<td>Maximum value for an object of type char(unsigned)</td>
</tr>
<tr>
<td>MB_LEN_MAX:</td>
<td>Maximum number of bytes in a multibyte character.</td>
</tr>
<tr>
<td>SHRT_MIN:</td>
<td>Minimum value for an object of type short int</td>
</tr>
<tr>
<td>SHRT_MAX:</td>
<td>Maximum value for an object of type short int</td>
</tr>
<tr>
<td>USHRT_MAX:</td>
<td>Maximum value for an object of type unsigned short int</td>
</tr>
</tbody>
</table>
INT_MIN: Minimum value for an object of type signed int
INT_MAX: Maximum value for an object of type signed int
UINT_MAX: Maximum value for an object of type unsigned int
LONG_MIN: Minimum value for an object of type signed long int
LONG_MAX: Maximum value for an object of type signed long int
ULONG_MAX: Maximum value for an object of type unsigned long int

locale.h

locale.h

locale.h (Localization not supported)
Iconv localization structure
SETLOCALE() returns null
LOCALCONV() returns locale

setjmp.h

setjmp.h

jmp_buf: An array used by the following functions
setjmp: Marks a return point for the next longjmp
longjmp: Jumps to the last marked point

stddef.h

stddef.h

ptrdiff_t: The basic type of a pointer
size_t: The type of the sizeof operator (int)
wchar_t: The type of the largest character set supported (char) (8 bits)
NULL A null pointer (0)
# stdio.h

<table>
<thead>
<tr>
<th>stdio.h</th>
<th>The standard error s stream (USE RS232 specified as stream or the first USE RS232)</th>
</tr>
</thead>
<tbody>
<tr>
<td>stderr</td>
<td>The standard output stream (USE RS232 specified as stream last USE RS232)</td>
</tr>
<tr>
<td>stdout</td>
<td>The standard input s stream (USE RS232 specified as stream last USE RS232)</td>
</tr>
</tbody>
</table>

# stdlib.h

<table>
<thead>
<tr>
<th>stdlib.h</th>
<th>structure type that contains two signed integers (quot and rem).</th>
</tr>
</thead>
<tbody>
<tr>
<td>div_t</td>
<td>structure type that contains two signed longs (quot and rem rem)</td>
</tr>
<tr>
<td>EXIT_FAILURE</td>
<td>returns 1</td>
</tr>
<tr>
<td>EXIT_SUCCESS</td>
<td>returns 0</td>
</tr>
<tr>
<td>RAND_MAX-</td>
<td></td>
</tr>
<tr>
<td>MBCUR_MAX-</td>
<td>1</td>
</tr>
<tr>
<td>SYSTEM()</td>
<td>Returns 0 (not supported)</td>
</tr>
<tr>
<td>Multibyte character and</td>
<td>Multibyte characters not supported</td>
</tr>
<tr>
<td>string functions:</td>
<td></td>
</tr>
<tr>
<td>MBLEN()</td>
<td>Returns the length of the string.</td>
</tr>
<tr>
<td>MBTOWC()</td>
<td>Returns 1.</td>
</tr>
<tr>
<td>WCTOMB()</td>
<td>Returns 1.</td>
</tr>
<tr>
<td>MBSTOWCS()</td>
<td>Returns length of string.</td>
</tr>
<tr>
<td>WBSTOMBS()</td>
<td>Returns length of string.</td>
</tr>
</tbody>
</table>

Stdlib.h functions included just for compliance with ANSI C.
SOFTWARE LICENSE AGREEMENT

Carefully read this Agreement prior to opening this package. By opening this package, you agree to abide by the following provisions. If you choose not to accept these provisions, promptly return the unopened package for a refund.

All materials supplied herein are owned by Custom Computer Services, Inc. (“CCS”) and is protected by copyright law and international copyright treaty. Software shall include, but not limited to, associated media, printed materials, and electronic documentation.

These license terms are an agreement between You (“Licensee” ) and CCS for use of the Software (“Software”). By installation, copy, download, or otherwise use of the Software, you agree to be bound by all the provisions of this License Agreement.

1. LICENSE - CCS grants Licensee a license to use in one of the two following options:
   1) Software may be used solely by single-user on multiple computer systems;
   2) Software may be installed on single-computer system for use by multiple users. Use of Software by additional users or on a network requires payment of additional fees.

   Licensee may transfer the Software and license to a third party; and such third party will be held to the terms of this Agreement. All copies of Software must be transferred to the third party or destroyed. Written notification must be sent to CCS for the transfer to be valid.
2. **APPLICATIONS SOFTWARE** - Use of this Software and derivative programs created by Licensee shall be identified as Applications Software, are not subject to this Agreement. Royalties are not be associated with derivative programs.

3. **WARRANTY** - CCS warrants the media to be free from defects in material and workmanship, and that the Software will substantially conform to the related documentation for a period of thirty (30) days after the date of purchase. CCS does not warrant that the Software will be free from error or will meet your specific requirements. If a breach in warranty has occurred, CCS will refund the purchase price or substitution of Software without the defect.

4. **LIMITATION OF LIABILITY AND DISCLAIMER OF WARRANTIES** – CCS and its suppliers disclaim any expressed warranties (other than the warranty contained in Section 3 herein), all implied warranties, including, but not limited to, the implied warranties of merchantability, of satisfactory quality, and of fitness for a particular purpose, regarding the Software.

Neither CCS, nor its suppliers, will be liable for personal injury, or any incidental, special, indirect or consequential damages whatsoever, including, without limitation, damages for loss of profits, loss of data, business interruption, or any other commercial damages or losses, arising out of or related to your use or inability to use the Software.

Licensee is responsible for determining whether Software is suitable for Applications.

©1994-2016 Custom Computer Services, Inc.
ALL RIGHTS RESERVED WORLDWIDE
PO BOX 2452
BROOKFIELD, WI 53008 U.S.A.