

CCS C Compiler Manual

PCD



May 2015

ALL RIGHTS RESERVED.
Copyright Custom Computer Services, Inc. 2015

Table of Contents

Overview	1
C Compiler.....	1
PCD.....	1
Technical Support.....	1
Directories	2
File Formats.....	2
Invoking the Command Line Compiler.....	4
PCW Overview	6
Menu.....	7
Editor Tabs	7
Slide Out Windows	7
Editor	8
Debugging Windows.....	8
Status Bar.....	9
Output Messages.....	9
Program Syntax	10
Overall Structure.....	10
Comment.....	10
Trigraph Sequences	12
Multiple Project Files.....	12
Multiple Compilation Units	13
Full Example Program	13
Statements	16
Statements	16
if.....	17
while	17
do-while	18
for	18
switch.....	19
return	19
goto.....	20
label	20
break.....	20
continue	21
expr.....	21
;.....	21
stmt.....	22
Expressions.....	23
Constants	23
Identifiers	24
Operators.....	24
Operator Precedence	26
Data Definitions.....	28
Data Definitions	28
Type Specifiers.....	29

Table of Contents

Type Qualifiers.....	30
Enumerated Types	31
Structures and Unions	31
typedef.....	33
Non-RAM Data Definitions.....	33
Using Program Memory for Data	35
Named Registers	37
PreProcessor	38
PRE-PROCESSOR DIRECTORY	38
__address__.....	40
__attribute_x.....	40
#asm #endasm #asm asis	41
#bank_dma.....	49
#bankx.....	50
#banky.....	50
#bit.....	51
__buildcount__	52
#build.....	52
#byte	53
#case	54
__date__	55
#define.....	55
definedinc	56
#device	57
__device__	59
#if expr #else #elif #endif	60
#error	61
#export (options).....	61
__file__	63
__filename__	63
#fill_rom	63
#fuses	64
#hexcomment	65
#id.....	65
#ifdef #ifndef #else #elif #endif	66
#ignore_warnings	67
#import (options).....	67
#include	68
#inline	69
#int_xxxx.....	70
__line__	73
#list	74
#line	74
#locate	75
#module.....	76
#nolist	77
#ocs	77
#opt.....	78

#org	78
#pin_select	80
__pcd__	84
#pragma	85
#profile	85
#recursive	86
#reserve	87
#rom	87
#separate	88
#serialize	89
#task	91
__time__	92
#type	92
#undef	94
_unicode	95
#use capture	96
#use delay	98
#use dynamic_memory	98
#use fast_io	99
#use fixed_io	99
#use i2c	100
#use profile()	101
#use pwm	102
#use rs232	104
#use rtos	108
#use spi	109
#use standard_io	111
#use timer	112
#use touchpad	113
#warning	114
#word	115
#zero_ram	116
Built-in Functions	117
BUILT-IN FUNCTIONS	117
abs()	125
sin() cos() tan() asin() acos() atan() sinh() cosh() tanh() atan2()	125
adc_done() adc_done2()	127
assert()	127
atoe	128
atof() atof48() atof64()	129
strtof48()	129
pin_select()	130
atoi() atol() atoi32()	131
atol32() atoi48() atoi64()	131
at_clear_interrupts()	132
at_disable_interrupts()	133
at_enable_interrupts()	133
at_get_capture()	134

Table of Contents

at_get_missing_pulse_delay()	135
at_get_period()	135
at_get_phase_counter()	136
at_get_resolution()	137
at_get_set_point()	137
at_get_set_point_error()	138
at_get_status()	138
at_interrupt_active()	139
at_set_compare_time()	140
at_set_missing_pulse_delay()	141
at_set_resolution()	141
at_set_set_point()	142
at_setup_cc()	143
bit_clear()	144
bit_first()	144
bit_last()	145
bit_set()	145
bit_test()	146
bsearch()	147
calloc()	148
ceil()	148
clear_interrupt()	149
cog_status()	149
cog_restart()	150
crc_calc()	150
crc_calc8()	150
crc_calc16()	150
crc_calc32()	150
crc_init(mode)	151
cwg_status()	152
cwg_restart()	153
dac_write()	153
dci_data_received()	154
dci_read()	154
dci_start()	155
dci_transmit_ready()	156
dci_write()	157
delay_cycles()	157
delay_ms()	158
delay_us()	159
disable_interrupts()	160
div()	161
ldiv()	161
dma_start()	162
dma_status()	163
enable_interrupts()	163
erase_program_memory	164
ext_int_edge()	165

fabs()	166
getc() getch() getchar() fgetc()	166
gets() fgets()	167
floor()	168
fmod()	168
printf() fprintf()	169
putc() putchar() fputc()	171
puts() fputs()	172
free()	172
frexp()	173
scanf()	174
get_capture()	176
get_capture()	177
get_capture_ccp1() get_capture_ccp2() get_capture_ccp3() get_capture_ccp4()) get_capture_ccp5()	177
get_capture32_ccp1() get_capture32_ccp2() get_capture32_ccp3() get_capture32_ccp4() get_capture32_ccp5()	179
get_capture_event()	180
get_capture_time()	181
get_capture32()	181
get_hspwm_capture()	182
get_motor_pwm_count()	183
get_nco_accumulator()	183
get_nco_inc_value()	184
get_ticks()	184
get_timerA()	185
get_timerB()	185
get_timerx()	186
get_timerxy()	187
get_timer_ccp1() get_timer_ccp2() get_timer_ccp3() get_timer_ccp4() get_timer_ccp5()	187
get_tris_x()	189
getc() getch() getchar() fgetc()	189
getenv()	190
gets() fgets()	195
goto_address()	196
high_speed_adc_done()	197
i2c_init()	198
i2c_isr_state()	198
i2c_poll()	199
i2c_read()	200
i2c_slaveaddr()	201
i2c_speed()	201
i2c_start()	202
i2c_stop()	203
i2c_write()	203
input()	204
input_change_x()	205

Table of Contents

input_state().....	206
input_x().....	206
interrupt_active().....	207
isalnum(char) isalpha(char)	208
iscntrl(x) isdigit(char).....	208
isgraph(x) islower(char) isspace(char) isupper(char) isxdigit(char) isprint(x) ispunct(x)	208
isamong().....	209
itoa()	210
kbhit().....	211
label_address().....	212
labs().....	212
ldexp().....	213
log()	213
log10()	214
longjmp().....	215
make8()	215
make16()	216
make32()	216
malloc().....	217
memcpy() memmove().....	218
memset()	218
modf()	219
_mul()	220
nargs()	220
offsetof() offsetofbit().....	221
output_x()	222
output_bit()	223
output_drive()	224
output_float()	224
output_high()	225
output_low().....	226
output_toggle().....	227
perror().....	227
pid_busy().....	228
pid_get_result().....	228
pid_read()	229
pid_write().....	230
pmp_address(address).....	231
pmp_output_full() pmp_input_full() pmp_overflow() pmp_error() pmp_timeout()	232
pmp_read()	233
pmp_write().....	234
port_x_pullups().....	235
pow() pwr()	236
printf() fprintf().....	236
profileout()	238
psp_output_full() psp_input_full() psp_overflow()	239

psp_read()	240
psp_write()	241
putc() putchar() fputc()	241
putc_send();	242
fputc_send();	242
puts() fputs()	243
pwm_off()	244
pwm_on()	244
pwm_set_duty()	245
pwm_set_duty_percent	245
pwm_set_frequency	246
qei_get_count()	247
qei_set_count()	247
qei_status()	248
qsort()	248
rand()	249
rcv_buffer_bytes()	250
rcv_buffer_full()	250
read_adc() read_adc2()	251
read_configuration_memory()	252
read_eeprom()	253
read_extended_ram()	253
read_program_memory()	254
read_high_speed_adc()	255
read_rom_memory()	257
read_sd_adc()	257
realloc()	258
release_io()	259
reset_cpu()	259
restart_cause()	260
restart_wdt()	260
rotate_left()	261
rotate_right()	262
rtc_alarm_read()	263
rtc_alarm_write()	263
rtc_read()	264
rtc_write()	265
rtos_await()	265
rtos_disable()	266
rtos_enable()	266
rtos_msg_poll()	267
rtos_msg_read()	267
rtos_msg_send()	268
rtos_overrun()	268
rtos_run()	269
rtos_signal()	270
rtos_stats()	270
rtos_terminate()	271

Table of Contents

rtos_wait()	271
rtos_yield()	272
set_adc_channel()	273
set_adc_channel2()	273
set_analog_pins()	273
scanf()	274
set_ccp1_compare_time() set_ccp2_compare_time() set_ccp3_compare_time() set_ccp4_compare_time() set_ccp5_compare_time()	277
set_cog_blanking()	278
set_cog_dead_band()	279
set_cog_phase()	280
set_compare_time()	280
set_hspwm_duty()	281
set_hspwm_event()	282
set_hspwm_override()	283
set_hspwm_phase()	283
set_motor_pwm_duty()	284
set_motor_pwm_event()	285
set_motor_unit()	285
set_nco_inc_value()	286
set_pullup()	287
set_pwm1_duty() set_pwm2_duty() set_pwm3_duty() set_pwm4_duty() set_pwm5_duty()	288
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	290
set_ticks()	291
setup_sd_adc_calibration()	291
set_sd_adc_channel()	292
set_timerA()	293
set_timerB()	293
set_timerx()	294
set_timerxy()	294
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	295
set_timer_ccp1() set_timer_ccp2() set_timer_ccp3() set_timer_ccp4() set_timer_ccp5()	296
set_timer_period_ccp1() set_timer_period_ccp2() set_timer_period_ccp3() set_timer_period_ccp4() set_timer_period_ccp5()	297
set_tris_x()	299
set_uart_speed()	299
setjmp()	300
setup_adc(mode)	301
setup_adc2(mode)	301
setup_adc_ports()	302
setup_adc_ports2()	302
setup_adc_reference()	303
setup_at()	303
setup_capture()	304

setup_ccp1() setup_ccp2() setup_ccp3() setup_ccp4() setup_ccp5()	
setup_ccp6().....	305
setup_clc1() setup_clc2() setup_clc3() setup_clc4().....	307
setup_comparator()	308
setup_compare().....	309
setup_crc(mode).....	309
setup_cog().....	310
setup_crc().....	311
setup_cwg()	312
setup_dac().....	313
setup_dci().....	314
setup_dma()	315
setup_high_speed_adc()	315
setup_high_speed_adc_pair()	316
setup_hspwm_blanking().....	317
setup_hspwm_chop_clock()	318
setup_hspwm_trigger().....	319
setup_hspwm_unit()	320
setup_hspwm()	321
setup_hspwm_unit_chop_clock().....	321
setup_low_volt_detect().....	323
setup_motor_pwm().....	323
setup_oscillator()	324
setup_pid().....	325
setup_pmp(option,address_mask).....	326
setup_power_pwm_pins().....	327
setup_psp(option,address_mask).....	328
setup_pwm1() setup_pwm2() setup_pwm3() setup_pwm4()	329
setup_qei().....	330
setup_rtc()	331
setup_rtc_alarm().....	331
setup_sd_adc().....	332
setup_smtx()	333
setup_spi() setup_spi2()	333
setup_timerx().....	334
setup_timer_A()	336
setup_timer_B()	336
setup_timer_0().....	337
setup_timer_1().....	338
setup_timer_2().....	338
setup_timer_3().....	339
setup_timer_4().....	340
setup_timer_5().....	341
setup_uart()	341
setup_vref()	342
setup_wdt().....	343
setup_zdc().....	344
shift_left().....	344

Table of Contents

shift_right().....	345
sleep().....	346
smtx_read()	347
smtx_reset_timer().....	348
smtx_start().....	349
smtx_status()	349
smtx_stop().....	350
smtx_write()	350
smtx_update().....	351
spi_data_is_in() spi_data_is_in2()	352
spi_init()	352
spi_prewrite(data);.....	353
spi_read() spi_read2()	353
spi_read3()	353
spi_read4()	353
spi_read_16()	354
spi_read2_16()	354
spi_read3_16()	354
spi_read4_16()	354
spi_speed	355
spi_write() spi_write2().....	356
spi_write3()	356
spi_write4()	356
spi_xfer().....	357
SPII_XFER_IN()	357
sprintf()	358
sqrt()	359
srand()	359
STANDARD STRING FUNCTIONS() memchr() memcmp() strcat() strchr()	
strcmp() strcoll() strcspn() strerror() stricmp() strlen() strlwr() strncat()	
strncmp() strncpy() strpbrk() strchr() strspn() strstr() strxfrm()	360
strcpy() strcpy()	362
strtod() strtodf() strtodf48().....	362
strtok()	363
strtol()	364
strtoul()	365
swap().....	366
tolower() toupper()	366
touchpad_getc().....	367
touchpad_hit().....	368
touchpad_state().....	369
tx_buffer_available()	370
tx_buffer_bytes().....	371
tx_buffer_full()	371
va_arg()	372
va_end()	373
va_start	374
write_configuration_memory()	374

PCD_May 2015

write_eeprom().....	375
write_extended_ram().....	376
write_program_memory()	377
zdc_status()	378
Standard C Include Files	379
errno.h	379
float.h.....	379
limits.h	380
locale.h	381
setjmp.h	381
stddef.h.....	381
stdio.h.....	381
stdlib.h	382
Software License Agreement	383
SOFTWARE LICENSE AGREEMENT	383

OVERVIEW

C Compiler

PCD Overview

Technical Support

Directories

File Formats

Invoking the Command Line Compiler

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

- .c This is the source file containing user C source code.
- .h These are standard or custom header files used to define pins, register, register bits,

functions and preprocessor directives.

- .pjt** This is the older pre- Version 5 project file which contains information related to the project.
- .ccspjt** This is the project file which contains information related to the project.
- 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>Project>Output Files

- .lst**
 - Mach code Includes the HEX opcode for each instruction
 - SFR names Instead of an address a name is used. For example instead of 044 is will show CORCON
 - Symbols Shows variable names instead of addresses
 - Interpret Adds a pseudo code interpretation to the right of assembly instruction to help understand the operation.

For example:

```
LSR W4, #8, W5 : W5=W4>>8
```

- .sym** This is the symbol map which shows each register location and what program variables are stored in each location.
- .sta** 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.
- .tre** 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.
- .hex** 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.
 - This is a binary containing machine code and debugging information.
- .cof** 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.
- .cod** This is a binary file containing debug information.
- .rtf** The output of the Documentation Generator is exported in a Rich Text File format which can be viewed using the RTF editor or Wordpad.
- .rvf** The Rich View Format is used by the RTF Editor within the IDE to view the Rich Text File.
- .dgr** The .DGR file is the output of the flowchart maker.
- .esym** 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.
- .xsym**
- .o** Relocatable object file
- .osym** 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.

.err Compiler error file

.ccsload used to link Windows 8 apps to CCSLoad

.ccssiow used to link Windows 8 apps to Serial Port Monitor

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)	-D	ate debug file
+FM	Select PCM (14 bit)	+DS	Standard .COD format debug file
+FH	Select PCH (PIC18XXX)	+DM	.MAP format debug file
+Yx	Optimization level x (0-9)	+DC	Expanded .COD format debug file
+FD	Select PCD (dsPIC30/dsPIC33/PIC24)	+DF	Enables the output of an COFF debug file.
+FS	Select SX (SX)	+EO	Old error file format
+ES	Standard error file	-T	Do not generate a tree file
+T	Create call tree (.TRE)	-A	Do not create stats file (.STA)
+A	Create stats file (.STA)	-EW	Suppress warnings (use with +EA)
+EW	Show warning messages	-E	Only show first error
+EA	Show all error messages and all warnings	+EX	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 file	+OWxxx	16-bit Intel HEX output 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	Keep compile status window up after compile
+Pxx	Keep status window up for xx seconds after compile
+PN	Keep status window up only if there are no errors
+PE	Keep status window up only if there are errors

+Z	Keep scratch files on disk after compile
+DF	COFF Debug file
I+="..."	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.

-P	Close compile window after compile is complete
+M	Generate a symbol file (.SYM)
-M	Do not create symbol file
+J	Create a project file (.PJT)
-J	Do not create PJT file
+ICD	Compile for use with an ICD
#xxx="yyy"	Set a global #define for id xxx with a value of yyy, example: #debug="true"
+Gxxx="yyy"	Same as #xxx="yyy"
+	
+?	Brings up a help file
-?	Same as +?
+STDOUT	Outputs errors to STDOUT (for use with third party editors)
+SETUP	Install CCSC into MPLAB (no compile is done)
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 +LNlst +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.

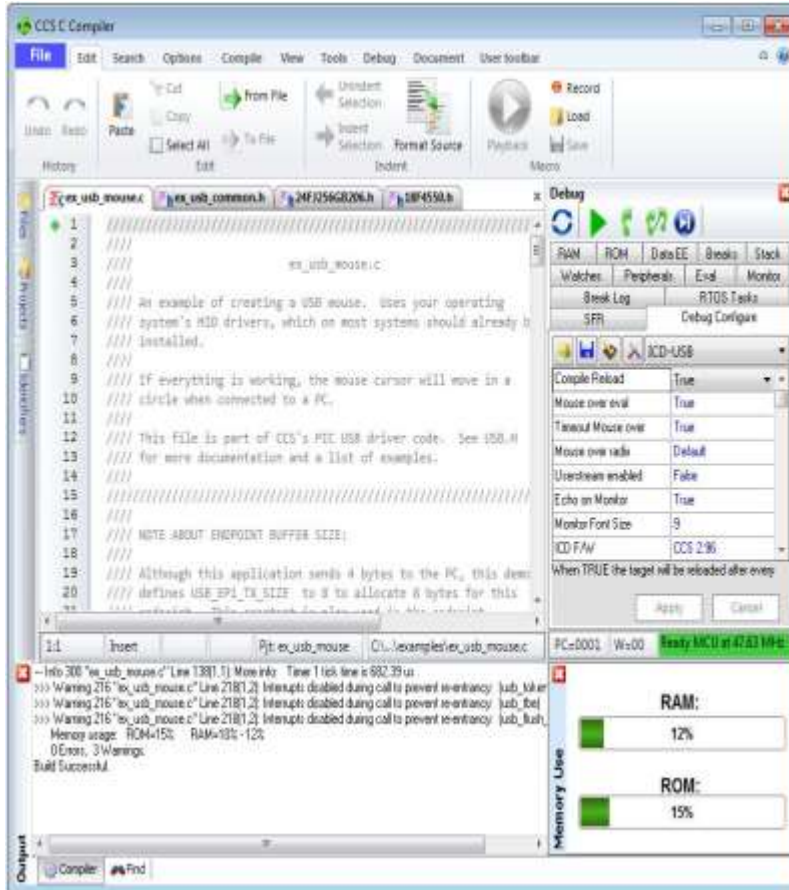
Examples:

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

PCW Overview

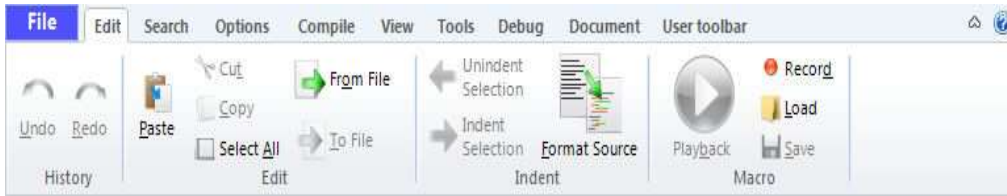
The PCW IDE provides the user an easy to use editor and environment for developing microcontroller applications. The IDE comprises of many components, which are summarized below. For more information and details, use the Help>PCW in the compiler..

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_CCS_PIC18F4550)
101      #include <18F4550.h>
102      #fuses HSPLL,NOWDT,NOPROTECT,NOLVP,NODEBUG,USBDIV,PLL5,CPUDIV1,VREGEN
103      #use delay(clock=48000000)
104
105      //leds ordered from bottom to top
106      #DEFINE LED1 PIN_A5 //green

```

Debug

Run Step Over Step Into Step Out

RAM	ROM	Data EE	Breaks	Stack
Watches	Peripherals	Eval	Monitor	
Break Log		RTOS Tasks		
SFR		Debug Configure		

ICD-USB

Compile Reload	True
Mouse over eval	True
Timeout Mouse over	True
Mouse over radix	Default
Userstream enabled	False
Echo on Monitor	True
Monitor Font Size	9
ICD F/W	CCS 2.96

When TRUE the target will be reloaded after every

Apply Cancel

PC=0001 W=00 Ready MCU at 47.63 MHz

Debugging Windows

Debugger control is done in the debugging windows. These windows allow you to 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.



PROGRAM SYNTAX

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:

```
/**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:

Sequence	Same as
??=	#
??([
??/	\
??)]
??'	^
??<	{
??!	
??>	}
??-	~

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:

```

main.c                #include <device header file>
                        #include<x.c>
                        #include<y.c>
                        #include <z.c>

x.c                   #include <x.h>
y.c                   #include <y.h>
z.c                   #include <z.h>

```

In this example there are 8 files and one compilation unit. `Main.c` is the only file compiled.

Note that the `#module` directive can be used in any include file to limit the visibility of the symbol in that file.

To separately compile your files see the section "multiple compilation units".

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, ///
/// comments that explains what
the program does, ///
/// and A/D samples over the RS-
232 interface. ///
////////////////////////////////////
////////////////////////////////////

#include <16F887.h>
//
preprocessor directive that

// selects the chip PIC16F887
#fuses NOPROTECT
// Code
protection turned off
#use delay(crystal=20mhz)
// preprocessor
directive that

// specifies the clock type
and speed

```

PCD_May 2015

```
#use rs232(baud=9600, xmit=PIN_C6,
rcv=PIN_C7) // preprocessor directive
that

// includes the rs232
libraries

void main() {
// main
function
int i, value, min, max;
// local variable
declaration
printf("Sampling:");
// printf
function included in the

// RS232 library
setup_port_a( ALL_ANALOG );
// A/D setup functions-
built-in
setup_adc( ADC_CLOCK_INTERNAL );
// Internal clock always
works
set_adc_channel( 0 );
// Set channel to
AN0
do {
/
/ do forever statement
min=255;

max=0;
for(i=0; i<=30; ++i) {
// Take 30 samples
delay_ms(100);
// Wait for a
tenth of a second
value = read_adc();
// A/D read functions-
built-in
if(value<min)
// Find smallest
sample
```

```
        min=value;
        if(value>max)
            // Find largest
sample
            max=value;
        }
        printf("\n\rMin: %2X  Max:
%2X\n\r",min,max);
    } while (TRUE);
}
```

STATEMENTS

Statements

STATEMENT

if (expr) stmt; [else stmt;]

while (expr) stmt;

do stmt while (expr);

for (expr1;expr2;expr3)
stmt;

switch (expr) {
case cexpr: stmt; //one or
more case [default:stmt]
... }

return [expr];

goto label;

label: stmt;

break;

continue;

expr;

;

{[stmt]}

Zero or more
declaration;

Example

```
if (x==25)
    x=0;
else
    x=x+1;
while (get_rtcc()!=0)
    putc('\n');
do {
    putc(c=getc());
} while (c!=0);
for (i=1;i<=10;++i)
    printf("%u\r\n",i);
switch (cmd) {
    case 0: printf("cmd 0");break;
    case 1: printf("cmd 1");break;
    default: printf("bad cmd");break;
}
return (5);
goto loop;
loop: i++;
break;
continue;
i=1;
;
{a=1;
 b=1;}
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)
    putchar('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

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

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

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

123	Decimal
123L	Forces type to & long (UL also allowed)
123LL	Forces type to &; 64 for PCD
0123	Octal
0x123	Hex
0b010010	Binary
123.456	Floating Point
123F	Floating Point (FL also allowed)
123.4E-5	Floating Point in scientific notation
'x'	Character
'010'	Octal Character
'xA5'	Hex Character
'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

- ABCDE 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.
- ID[X] Single Subscript
- ID[X][X] Multiple Subscripts
- ID.ID Structure or union reference
- ID->ID Structure or union reference

Operators

- + Addition Operator
- += Addition assignment operator, $x+=y$, is the same as $x=x+y$
- [] Array subscript operator
- &= Bitwise and assignment operator, $x&=y$, is the same as $x=x&y$
- & Address operator
- & Bitwise and operator
- ^= Bitwise exclusive or assignment operator, $x^=y$, is the same as $x=x^y$

<code>^</code>	Bitwise exclusive or operator
<code> =</code>	Bitwise inclusive or assignment operator, <code>x =y</code> , is the same as <code>x=x y</code>
<code> </code>	Bitwise inclusive or operator
<code>?:</code>	Conditional Expression operator
<code>--</code>	Decrement
<code>/=</code>	Division assignment operator, <code>x/=y</code> , is the same as <code>x=x/y</code>
<code>/</code>	Division operator
<code>==</code>	Equality
<code>></code>	Greater than operator
<code>>=</code>	Greater than or equal to operator
<code>++</code>	Increment
<code>*</code>	Indirection operator
<code>!=</code>	Inequality
<code><<=</code>	Left shift assignment operator, <code>x<<=y</code> , is the same as <code>x=x<<y</code>
<code><</code>	Less than operator
<code><<</code>	Left Shift operator
<code><=</code>	Less than or equal to operator
<code>&&</code>	Logical AND operator
<code>!</code>	Logical negation operator
<code> </code>	Logical OR operator
<code>.</code>	Member operator for structures and unions
<code>%=</code>	Modules assignment operator <code>x%=y</code> , is the same as <code>x=x%y</code>
<code>%</code>	Modules operator
<code>*=</code>	Multiplication assignment operator, <code>x*=y</code> , is the same as <code>x=x*y</code>
<code>*</code>	Multiplication operator
<code>~</code>	One's complement operator
<code>>>=</code>	Right shift assignment, <code>x>>=y</code> , is the same as <code>x=x>>y</code>

>>	Right shift operator
->	Structure Pointer operation
-=	Subtraction assignment operator, x-=y, is the same as x=x- y
-	Subtraction operator
sizeof	Determines size in bytes of operand

See also: Operator Precedence

Operator Precedence

PIN DESCENDING PRECEDENCE

(expr)	expr++	expr->expr	expr.expr	Associativity Left to Right
++expr	expr++	- -expr	expr - -	Left to Right
!expr	~expr	+expr	-expr	Right to Left
(type)expr	*expr	&value	sizeof(type)	Right to Left
expr*expr	expr/expr	expr%expr		Left to Right
expr+expr	expr-expr			Left to Right
expr<<expr	expr>>expr			Left to Right
expr<expr	expr<=expr	expr>expr	expr>=expr	Left to Right
expr==expr	expr!=expr			Left to Right
expr&expr				Left to Right
expr^expr				Left to Right
expr expr				Left to Right
expr&& expr				Left to Right
expr expr				Left to Right
expr ? expr: expr				Right to Left
lvalue = expr	lvalue+=expr	lvalue-=expr		Right to Left
lvalue *=expr	lvalue/=expr	lvalue%=expr		Right to Left

Expressions

lvalue>>=expr lvalue<<=exp
r lvalue&=expr

Right
to Left

lvalue^=expr lvalue|=expr

Right
to Left

expr, expr

Left to Right

(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:

```
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:

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

SEE ALSO:

Type Specifiers/ Basic Types

Type Qualifiers

Enumerated Types

Structures & Unions

typedef

Named Registers

Type Specifiers

Basic Types

Type-Specifier	Size	Range		Digits
		Unsigned	Signed	
int1	1 bit number	0 to 1	N/A	1/2
int8	8 bit number	0 to 255	-128 to 127	2-3
int16	16 bit number	0 to 65535	-32768 to 32767	4-5
int32	32 bit number	0 to 4294967295	-2147483648 to 2147483647	9-10
int48	48 bit number	0 to 281474976710655	-140737488355328 to 140737488355327	14-15
int64	64 bit number	N/A	-9223372036854775808 to 9223372036854775807	18-19
float32	32 bit float	-1.5 x 10 ⁴⁵ to 3.4 x 10 ³⁸		7-8
float48	48 bit float (higher precision)	-2.9 x 10 ³⁹ to 1.7 x 10 ³⁸		11-12
float64	64 bit float	-5.0 x 10 ³²⁴ to 1.7 x 10 ³⁰⁸		15-16

C Standard Type

short
char
int
long
long long
float
double

Default Type

signed int8
signed int8
signed int16
signed int32
signed int64
float32

Note: All types by default are 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

Type-Qualifier

static	Variable is globally active and initialized to 0. Only accessible from this compilation unit.
auto	Variable exists only while the procedure is active. This is the default and AUTO need not be used.
double	Is a reserved word but is not a supported data type.
extern	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.
register	If possible a CPU register instead of a RAM location.
_fixed(n)	Creates a fixed point decimal number where <i>n</i> is how many decimal places to implement.
unsigned	Data is always positive.
signed	Data can be negative or positive. This is the default data type if not specified.
volatile	Tells the compiler optimizer that this variable can be changed at any point during execution.
const	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=)
rom	Forces data into program memory. Pointers may be used to this data but they can not be mixed with RAM pointers.
roml	Same as rom except only the even program memory locations are used.


void	Built-in basic type. Type void is used to indicate no specific type in places where a type is required.
readonly	Writes to this variable should be dis-allowed
_bif	Used for compiler built in function prototypes on the same line
__attribute__	Sets various attributes

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

Enumerated Types

enum enumeration type: creates a list of integer constants.

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



One or more comma separated

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
```

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

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]
```



One or more,
semi-colon
separated



Zero
or more

For example:


```
struct data_record {  
    int  a[2];  
    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
```


Field Allocation

- Fields are allocated in the order they appear.
- The low bits of a byte are filled first.
- Fields 16 bits and up are aligned to an even byte boundary. Some Bits may be unused.
- No Field will span from an odd byte to an even byte unless the field width is a multiple of 16 bits.

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.

```
union[*] [id] {           type-qualifier [*] id           [:bits];           } [id]
```

 One or more,
semi-colon
separated

 Zero
or more

For example:

```
union u_tag {  
    int  ival;  
    long lval;  
    float fval;  
}; //u_tag is a union type that can  
   hold a float
```

SEE ALSO: Declarations, Type Specifiers, Type Qualifiers, Enumerated Types, typedef, Named Registers

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.

typedef [type-qualifier] [type-specifier] [declarator];

For example:

```
typedef int mybyte;           // mybyte can be used in declaration
                             to
                             // specify the int type
typedef short mybit;        // mybyte can be used in declaration
                             to
                             // specify the int type
typedef enum {red,          //colors can be used to declare
green=2,blue}colors;      //variable of this enum type
```

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

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 :

```
#type default=addressmodname // all the variable
declarations that           // follow will use

this memory region
#type default=              // goes back to the
default mode
```

For example:

```
Type default=emi           //emi is the
addressmod name defined
char buffer[8192];
#include <memoryhog.h>
#type default=
```

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
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:

```
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:

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

Where n is optional and id is the table identifier.

For example:

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

#ROM directive:

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

The syntax is:

```
#rom address = {data, data, ... , data}
```

For example:

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

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

Places null terminated string in ROM

```
#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:

- `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
 - `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:

```
register _name type id;  
Or  
register constant type id;
```

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

Examples:

```
register _status int8 status_reg;  
register _T1IF int8 timer_interrupt;  
register 0x04 int16 file_select_register;
```

PREPROCESSOR

PRE-PROCESSOR DIRECTORY

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

```
#INLINE  
#PRAGMA INLINE
```

_address	40
_attribute_x	40
#asm #endasm #asm asis	41
#bank_dma	49
#bankx	50
#banky	50
#bit	51
_buildcount	52
#build	52
#byte	53
#case	54
_date	55
#define	55
definedinc	56
#device	57
_device	59
#if expr #else #elif #endif	60
#error	61
#export (options)	61
_file	63
_filename	63
#fill_rom	63
#fuses	64
#hexcomment	65

<u>#id</u>	65
<u>#ifdef #ifndef #else #elif #endif</u>	66
<u>#ignore_warnings</u>	67
<u>#import (options)</u>	67
<u>#include</u>	68
<u>#inline</u>	69
<u>#int xxxx</u>	70
<u>line</u>	73
<u>#list</u>	74
<u>#line</u>	74
<u>#locate</u>	75
<u>#module</u>	76
<u>#nolist</u>	77
<u>#ocs</u>	77
<u>#opt</u>	78
<u>#org</u>	78
<u>#pin_select</u>	80
<u>pcd</u>	84
<u>#pragma</u>	85
<u>#profile</u>	85
<u>#recursive</u>	86
<u>#reserve</u>	87
<u>#rom</u>	87
<u>#separate</u>	88
<u>#serialize</u>	89
<u>#task</u>	91
<u>time</u>	92
<u>#type</u>	92
<u>#undef</u>	94
<u>unicode</u>	95
<u>#use capture</u>	96
<u>#use delay</u>	98
<u>#use dynamic_memory</u>	98
<u>#use fast_io</u>	99
<u>#use fixed_io</u>	99
<u>#use i2c</u>	100
<u>#use profile()</u>	101
<u>#use pwm</u>	102
<u>#use rs232</u>	104
<u>#use rtos</u>	108
<u>#use spi</u>	109
<u>#use standard_io</u>	111
<u>#use timer</u>	112
<u>#use touchpad</u>	113
<u>#warning</u>	114
<u>#word</u>	115
<u>#zero_ram</u>	116

__address__

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

For example:

```
__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:

```
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:

```
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:

```
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: <pre>int8 array[256] __attribute__((aligned(0x1000)));</pre> 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:	<pre>struct __attribute__((packed)) { int8 a; int8 b; } test; int8 array[256] __attribute__((aligned(0x1000)));</pre>
Example Files:	None

#asm #endasm #asm asis

Syntax:	#ASM or #ASM ASIS code #ENDASM
Elements:	code is a list of assembly language instructions
Examples:	<pre>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 }</pre>
Example Files:	FFT.c

Also See: None

ADD	Wa,Wb,Wd	Wd = Wa+ Wb
ADD	f,W	W0 = f+Wd
ADD	lit10,Wd	Wd = lit10+Wd
ADD	Wa,lit5,Wd	Wd = lit5+Wa
ADD	f,F	f = f+Wd
ADD	acc	Acc = AccA+AccB
ADD	Wd,{lit4},acc	Acc = Acc+(Wa shifted slit4)
ADD.B	lit10,Wd	Wd = lit10+Wd (byte)
ADD	Wd,{lit4},acc	Acc = Acc+(Wa shifted slit4)
ADD.B	lit10,Wd	Wd = lit10+Wd (byte)
ADD.B	f,F	f = f+Wd (byte)
ADD.B	Wa,Wb,Wd	Wd = Wa+Wb (byte)
ADD.B	Wa,lit5,Wd	Wd = lit5+Wa (byte)
ADD.B	f,W	W0 = f+Wd (byte)
ADDC	f,W	W d = f + W a + C
ADDC	lit10,Wd	Wd = lit10+Wd+C
ADDC	Wa,lit5,Wd	Wd = lit5+Wa+C
ADDC	f,F	Wd = f+Wa+C
ADDC	Wa,Wb,Wd	Wd = Wa+Wb+C
ADDC.B	lit10,Wd	Wd = lit10+Wd+C (byte)
ADDC.B	Wa,Wb,Wd	Wd = Wa+Wb+C (byte)
ADDC.B	Wa,lit5,Wd	Wd = lit5+Wa+C (byte)
ADDC.B	f,W	Wd = f+Wa+C (byte)
ADDC.B	f,F	Wd = f+Wa+C (byte)
AND	Wa,Wb,Wd	Wd = Wa.&Wb
AND	lit10,Wd	Wd = lit10.&Wd

AND	f,W	W0 = f.&.Wa
AND	f,F	f = f.&.Wa
AND	Wa,lit5,Wd	Wd = lit5.&.Wa
AND.B	f,W	W0 = f.&.Wa (byte)
AND.B	Wa,Wb,Wd	Wd = Wa.&.Wb (byte)
AND.B	lit10,Wd	Wd = lit10.&.Wd (byte)
AND.B	f,F	f = f.&.Wa (byte)
AND.B	Wa,lit5,Wd	Wd = lit5.&.Wa (byte)
ASR	f,W	W0 = f >> 1 arithmetic
ASR	f,F	f = f >> 1 arithmetic
ASR	Wa,Wd	Wd = Wa >> 1 arithmetic
ASR	Wa,lit4,Wd	Wd = Wa >> lit4 arithmetic
ASR	Wa,Wb,Wd	Wd = Wa >> Wb arithmetic
ASR.B	f,F	f = f >> 1 arithmetic (byte)
ASR.B	f,W	W0 = f >> 1 arithmetic (byte)
ASR.B	Wa,Wd	Wd = Wa >> 1 arithmetic (byte)
BCLR	f,B	f.bit = 0
BCLR	Wd,B	Wa.bit = 0
BCLR.B	Wd,B	Wa.bit = 0 (byte)
BRA	a	Branch unconditionally
BRA	Wd	Branch PC+Wa
BRA BZ	a	Branch if Zero
BRA C	a	Branch if Carry (no borrow)
BRA GE	a	Branch if greater than or equal
BRA GEU	a	Branch if unsigned greater than or equal
BRA GT	a	Branch if greater than
BRA GTU	a	Branch if unsigned greater than
BRA LE	a	Branch if less than or equal
BRA LEU	a	Branch if unsigned less than or equal
BRA LT	a	Branch if less than
BRA LTU	a	Branch if unsigned less than
BRA N	a	Branch if negative
BRA NC	a	Branch if not carry (Borrow)
BRA NN	a	Branch if not negative
BRA NOV	a	Branch if not Overflow
BRA NZ	a	Branch if not Zero
BRA OA	a	Branch if Accumulator A overflow
BRA OB	a	Branch if Accumulator B overflow
BRA OV	a	Branch if Overflow
BRA SA	a	Branch if Accumulator A Saturate
BRA SB	a	Branch if Accumulator B Saturate
BRA Z	a	Branch if Zero
BREAK		ICD Break
BSET	Wd,B	Wa.bit = 1
BSET	f,B	f.bit = 1
BSET.B	Wd,B	Wa.bit = 1 (byte)
BSW.C	Wa,Wd	Wa.Wb = C
BSW.Z	Wa,Wd	Wa.Wb = Z

BTG	Wd,B	Wa.bit = ~Wa.bit
BTG	f,B	f.bit = ~f.bit
BTG.B	Wd,B	Wa.bit = ~Wa.bit (byte)
BTSC	f,B	Skip if f.bit = 0
BTSC	Wd,B	Skip if Wa.bit4 = 0
BTSS	f,B	Skip if f.bit = 1
BTSS	Wd,B	Skip if Wa.bit = 1
BTST	f,B	Z = f.bit
BTST.C	Wa,Wd	C = Wa.Wb
BTST.C	Wd,B	C = Wa.bit
BTST.Z	Wd,B	Z = Wa.bit
BTST.Z	Wa,Wd	Z = Wa.Wb
BTSTS	f,B	Z = f.bit; f.bit = 1
BTSTS.C	Wd,B	C = Wa.bit; Wa.bit = 1
BTSTS.Z	Wd,B	Z = Wa.bit; Wa.bit = 1
CALL	a	Call subroutine
CALL	Wd	Call [Wa]
CLR	f,F	f = 0
CLR	acc,da,dc,pi	Acc = 0; prefetch=0
CLR	f,W	W0 = 0
CLR	Wd	Wd = 0
CLR.B	f,W	W0 = 0 (byte)
CLR.B	Wd	Wd = 0 (byte)
CLR.B	f,F	f = 0 (byte)
CLRWDT		Clear WDT
COM	f,F	f = ~f
COM	f,W	W0 = ~f
COM	Wa,Wd	Wd = ~Wa
COM.B	f,W	W0 = ~f (byte)
COM.B	Wa,Wd	Wd = ~Wa (byte)
COM.B	f,F	f = ~f (byte)
CP	W,f	Status set for f - W0
CP	Wa,Wd	Status set for Wb $\hat{=}$ Wa
CP	Wd,lit5	Status set for Wa $\hat{=}$ lit5
CP.B	W,f	Status set for f - W0 (byte)
CP.B	Wa,Wd	Status set for Wb $\hat{=}$ Wa (byte)
CP.B	Wd,lit5	Status set for Wa $\hat{=}$ lit5 (byte)
CP0	Wd	Status set for Wa $\hat{=}$ 0
CP0	W,f	Status set for f $\hat{=}$ 0
CP0.B	Wd	Status set for Wa $\hat{=}$ 0 (byte)
CP0.B	W,f	Status set for f $\hat{=}$ 0 (byte)
CPB	Wd,lit5	Status set for Wa $\hat{=}$ lit5 $\hat{=}$ C
CPB	Wa,Wd	Status set for Wb $\hat{=}$ Wa $\hat{=}$ C
CPB	W,f	Status set for f $\hat{=}$ W0 - C
CPB.B	Wa,Wd	Status set for Wb $\hat{=}$ Wa $\hat{=}$ C (byte)
CPB.B	Wd,lit5	Status set for Wa $\hat{=}$ lit5 $\hat{=}$ C (byte)
CPB.B	W,f	Status set for f $\hat{=}$ W0 - C (byte)
CPSEQ	Wa,Wd	Skip if Wa = Wb

CPSEQ.B	Wa,Wd	Skip if Wa = Wb (byte)
CPSGT	Wa,Wd	Skip if Wa > Wb
CPSGT.B	Wa,Wd	Skip if Wa > Wb (byte)
CPSLT	Wa,Wd	Skip if Wa < Wb
CPSLT.B	Wa,Wd	Skip if Wa < Wb (byte)
CPSNE	Wa,Wd	Skip if Wa != Wb
CPSNE.B	Wa,Wd	Skip if Wa != Wb (byte)
DAW.B	Wd	Wa = decimal adjust Wa
DEC	Wa,Wd	Wd = Wa - 1
DEC	f,W	W0 = f - 1
DEC	f,F	f = f - 1
DEC.B	f,F	f = f - 1 (byte)
DEC.B	f,W	W0 = f - 1 (byte)
DEC.B	Wa,Wd	Wd = Wa - 1 (byte)
DEC2	Wa,Wd	Wd = Wa - 2
DEC2	f,W	W0 = f - 2
DEC2	f,F	f = f - 2
DEC2.B	Wa,Wd	Wd = Wa - 2 (byte)
DEC2.B	f,W	W0 = f - 2 (byte)
DEC2.B	f,F	f = f - 2 (byte)
DISI	lit14	Disable Interrupts lit14 cycles
DIV.S	Wa,Wd	Signed 16/16-bit integer divide
DIV.SD	Wa,Wd	Signed 16/16-bit integer divide (dword)
DIV.U	Wa,Wd	Unsigned 16/16-bit integer divide
DIV.UD	Wa,Wd	Unsigned 16/16-bit integer divide (dword)
DIVF	Wa,Wd	Signed 16/16-bit fractional divide
DO	lit14,a	Do block lit14 times
DO	Wd,a	Do block Wa times
ED	Wd*Wd,acc,da,db	Euclidean Distance (No Accumulate)
EDAC	Wd*Wd,acc,da,db	Euclidean Distance
EXCH	Wa,Wd	Swap Wa and Wb
FBCL	Wa,Wd	Find bit change from left (Msb) side
FEX		ICD Execute
FF1L	Wa,Wd	Find first one from left (Msb) side
FF1R	Wa,Wd	Find first one from right (Lsb) side
GOTO	a	GoTo
GOTO	Wd	GoTo [Wa]
INC	f,W	W0 = f + 1
INC	Wa,Wd	Wd = Wa + 1
INC	f,F	f = f + 1
INC.B	Wa,Wd	Wd = Wa + 1 (byte)
INC.B	f,F	f = f + 1 (byte)
INC.B	f,W	W0 = f + 1 (byte)
INC2	f,W	W0 = f + 2
INC2	Wa,Wd	Wd = Wa + 2
INC2	f,F	f = f + 2
INC2.B	f,W	W0 = f + 2 (byte)
INC2.B	f,F	f = f + 2 (byte)

INC2.B	Wa,Wd	Wd = Wa + 2 (byte)
IOR	lit10,Wd	Wd = lit10 Wd
IOR	f,F	f = f Wa
IOR	f,W	W0 = f Wa
IOR	Wa,lit5,Wd	Wd = Wa. lit5
IOR	Wa,Wb,Wd	Wd = Wa. .Wb
IOR.B	Wa,Wb,Wd	Wd = Wa. .Wb (byte)
IOR.B	f,W	W0 = f Wa (byte)
IOR.B	lit10,Wd	Wd = lit10 Wd (byte)
IOR.B	Wa,lit5,Wd	Wd = Wa. .lit5 (byte)
IOR.B	f,F	f = f Wa (byte)
LAC	Wd,{lit4},acc	Acc = Wa shifted slit4
LNK	lit14	Allocate Stack Frame
LSR	f,W	W0 = f >> 1
LSR	Wa,lit4,Wd	Wd = Wa >> lit4
LSR	Wa,Wd	Wd = Wa >> 1
LSR	f,F	f = f >> 1
LSR	Wa,Wb,Wd	Wd = Wb >> Wa
LSR.B	f,W	W0 = f >> 1 (byte)
LSR.B	f,F	f = f >> 1 (byte)
LSR.B	Wa,Wd	Wd = Wa >> 1 (byte)
MAC	Wd*Wd,acc,da,dc	Acc = Acc + Wa * Wa; {prefetch}
MAC	Wd*Wc,acc,da,dc,pi	Acc = Acc + Wa * Wb; {[W13] = Acc}; {prefetch}
MOV	W,f	f = Wa
MOV	f,W	W0 = f
MOV	f,F	f = f
MOV	Wd,?	F = Wa
MOV	Wa+lit,Wd	Wd = [Wa +Slit10]
MOV	?,Wd	Wd = f
MOV	lit16,Wd	Wd = lit16
MOV	Wa,Wd	Wd = Wa
MOV	Wa,Wd+lit	[Wd + Slit10] = Wa
MOV.B	lit8,Wd	Wd = lit8 (byte)
MOV.B	W,f	f = Wa (byte)
MOV.B	f,W	W0 = f (byte)
MOV.B	f,F	f = f (byte)
MOV.B	Wa+lit,Wd	Wd = [Wa +Slit10] (byte)
MOV.B	Wa,Wd+lit	[Wd + Slit10] = Wa (byte)
MOV.B	Wa,Wd	Wd = Wa (byte)
MOV.D	Wa,Wd	Wd:Wd+1 = Wa:Wa+1
MOV.D	Wa,Wd	Wd:Wd+1 = Wa:Wa+1
MOVSAC	acc,da,dc,pi	Move ? to ? and ? To ?
MPY	Wd*Wc,acc,da,dc	Acc = Wa*Wb
MPY	Wd*Wd,acc,da,dc	Square to Acc
MPY.N	Wd*Wc,acc,da,dc	Acc = -(Wa*Wb)
MSC	Wd*Wc,acc,da,dc,pi	Acc = Acc â€• Wa*Wb
MUL	W,f	W3:W2 = f * Wa
MUL.B	W,f	W3:W2 = f * Wa (byte)

MUL.SS	Wa,Wd	{Wd+1,Wd}= sign(Wa) * sign(Wb)
MUL.SU	Wa,Wd	{Wd+1,Wd} = sign(Wa) * unsign(Wb)
MUL.SU	Wa,lit5,Wd	{Wd+1,Wd}= sign(Wa) * unsign(lit5)
MUL.US	Wa,Wd	{Wd+1,Wd} = unsign(Wa) * sign(Wb)
MUL.UU	Wa,Wd	{Wd+1,Wd} = unsign(Wa) * unsign(Wb)
MUL.UU	Wa,lit5,Wd	{Wd+1,Wd} = unsign(Wa) * unsign(lit5)
NEG	f,F	f = - f
PUSH	Wd	Push Wa to TOS
PUSH.D	Wd	PUSH double Wa:Wa + 1 to TOS
PUSH.S		PUSH shadow registers
PWRSAB	lit1	Enter Power-saving mode lit1
RCALL	a	Call (relative)
RCALL	Wd	Call Wa
REPEAT	lit14	Repeat next instruction (lit14 + 1) times
REPEAT	Wd	Repeat next instruction (Wa + 1) times
RESET		Reset
RETFIE		Return from interrupt enable
RETLW	lit10,Wd	Return; Wa = lit10
RETLW.B	lit10,Wd	Return; Wa = lit10 (byte)
RETURN		Return
RLC	Wa,Wd	Wd = rotate left through Carry Wa
RLC	f,F	f = rotate left through Carry f
RLC	f,W	W0 = rotate left through Carry f
RLC.B	f,F	f = rotate left through Carry f (byte)
RLC.B	f,W	W0 = rotate left through Carry f (byte)
RLC.B	Wa,Wd	Wd = rotate left through Carry Wa (byte)
RLNC	Wa,Wd	Wd = rotate left (no Carry) Wa
RLNC	f,F	f = rotate left (no Carry) f
RLNC	f,W	W0 = rotate left (no Carry) f
RLNC.B	f,W	W0 = rotate left (no Carry) f (byte)
RLNC.B	Wa,Wd	Wd = rotate left (no Carry) Wa (byte)
RLNC.B	f,F	f = rotate left (no Carry) f (byte)
RRC	f,F	f = rotate right through Carry f
RRC	Wa,Wd	Wd = rotate right through Carry Wa
RRC	f,W	W0 = rotate right through Carry f
RRC.B	f,W	W0 = rotate right through Carry f (byte)
RRC.B	f,F	f = rotate right through Carry f (byte)
RRC.B	Wa,Wd	Wd = rotate right through Carry Wa (byte)
RRNC	f,F	f = rotate right (no Carry) f
RRNC	f,W	W0 = rotate right (no Carry) f
RRNC	Wa,Wd	Wd = rotate right (no Carry) Wa
RRNC.B	f,F	f = rotate right (no Carry) f (byte)
RRNC.B	Wa,Wd	Wd = rotate right (no Carry) Wa (byte)
RRNC.B	f,W	W0 = rotate right (no Carry) f (byte)
SAC	acc,{lit4},Wd	Wd = Acc slit 4
SAC.R	acc,{lit4},Wd	Wd = Acc slit 4 with rounding
SE	Wa,Wd	Wd = sign-extended Wa
SETM	Wd	Wd = 0xFFFF

SETM	f,F	W0 = 0xFFFF
SETM.B	Wd	Wd = 0xFFFF (byte)
SETM.B	f,W	W0 = 0xFFFF (byte)
SETM.B	f,F	W0 = 0xFFFF (byte)
SFTAC	acc,Wd	Arithmetic shift Acc by (Wa)
SFTAC	acc,lit5	Arithmetic shift Acc by Slit6
SL	f,W	W0 = f << 1
SL	Wa,Wb,Wd	Wd = Wa << Wb
SL	Wa,lit4,Wd	Wd = Wa << lit4
SL	Wa,Wd	Wd = Wa << 1
SL	f,F	f = f << 1
SL.B	f,W	W0 = f << 1 (byte)
SL.B	Wa,Wd	Wd = Wa << 1 (byte)
SL.B	f,F	f = f << 1 (byte)
SSSTEP		ICD Single Step
SUB	f,F	f = f â€“ W0
SUB	f,W	W0 = f â€“ W0
SUB	Wa,Wb,Wd	Wd = Wa â€“ Wb
SUB	Wa,lit5,Wd	Wd = Wa â€“ lit5
SUB	acc	Acc = AccA â€“ AccB
SUB	lit10,Wd	Wd = Wd â€“ lit10
SUB.B	Wa,lit5,Wd	Wd = Wa â€“ lit5 (byte)
SUB.B	lit10,Wd	Wd = Wd â€“ lit10 (byte)
SUB.B	f,W	W0 = f â€“ W0 (byte)
SUB.B	Wa,Wb,Wd	Wd = Wa â€“ Wb (byte)
SUB.B	f,F	f = f â€“ W0 (byte)
SUBB	f,W	W0 = f â€“ W0 â€“ C
SUBB	Wa,Wb,Wd	Wd = Wa â€“ Wb â€“ C
SUBB	f,F	f = f â€“ W0 â€“ C
SUBB	Wa,lit5,Wd	Wd = Wa â€“ lit5 - C
SUBB	lit10,Wd	Wd = Wd â€“ lit10 â€“ C
SUBB.B	lit10,Wd	Wd = Wd â€“ lit10 â€“ C (byte)
SUBB.B	Wa,Wb,Wd	Wd = Wa â€“ Wb â€“ C (byte)
SUBB.B	f,F	f = f â€“ W0 â€“ C (byte)
SUBB.B	Wa,lit5,Wd	Wd = Wa â€“ lit5 - C (byte)
SUBB.B	f,W	W0 = f â€“ W0 â€“ C (byte)
SUBBR	Wa,lit5,Wd	Wd = lit5 â€“ Wa - C
SUBBR	f,W	W0 = W0 â€“ f â€“ C
SUBBR	f,F	f = W0 â€“ f â€“ C
SUBBR	Wa,Wb,Wd	Wd = Wa â€“ Wb - C
SUBBR.B	f,F	f = W0 â€“ f â€“ C (byte)
SUBBR.B	f,W	W0 = W0 â€“ f â€“ C (byte)
SUBBR.B	Wa,Wb,Wd	Wd = Wa â€“ Wb - C (byte)
SUBBR.B	Wa,lit5,Wd	Wd = lit5 â€“ Wa - C (byte)
SUBR	Wa,lit5,Wd	Wd = lit5 â€“ Wb
SUBR	f,F	f = W0 â€“ f
SUBR	Wa,Wb,Wd	Wd = Wa â€“ Wb
SUBR	f,W	W0 = W0 â€“ f

SUBR.B	Wa,Wb,Wd	Wd = Wa $\hat{=}$ Wb (byte)
SUBR.B	f,F	f = W0 $\hat{=}$ f (byte)
SUBR.B	Wa,lit5,Wd	Wd = lit5 $\hat{=}$ Wb (byte)
SUBR.B	f,W	W0 = W0 $\hat{=}$ f (byte)
SWAP	Wd	Wa = byte or nibble swap Wa
SWAP.B	Wd	Wa = byte or nibble swap Wa (byte)
TBLRDH	Wa,Wd	Wd = ROM[Wa] for odd ROM
TBLRDH.B	Wa,Wd	Wd = ROM[Wa] for odd ROM (byte)
TBLRDL	Wa,Wd	Wd = ROM[Wa] for even ROM
TBLRDL.B	Wa,Wd	Wd = ROM[Wa] for even ROM (byte)
TBLWTH	Wa,Wd	ROM[Wa] = Wd for odd ROM
TBLWTH.B	Wa,Wd	ROM[Wa] = Wd for odd ROM (byte)
TBLWTL	Wa,Wd	ROM[Wa] = Wd for even ROM
TBLWTL.B	Wa,Wd	ROM[Wa] = Wd for even ROM (byte)
ULNK		Deallocate Stack Frame
URUN		ICD Run
XOR	Wa,Wb,Wd	Wd = Wa \wedge Wb
XOR	f,F	f = f \wedge W0
XOR	f,W	W0 = f \wedge W0
XOR	Wa,lit5,Wd	Wd = Wa \wedge lit5
XOR	lit10,Wd	Wd = Wd \wedge lit10
XOR.B	lit10,Wd	Wd = Wd \wedge lit10 (byte)
XOR.B	f,W	W0 = f \wedge W0 (byte)
XOR.B	Wa,lit5,Wd	Wd = Wa \wedge lit5 (byte)
XOR.B	Wa,Wb,Wd	Wd = Wa \wedge Wb (byte)
XOR.B	f,F	f = f \wedge W0 (byte)
ZE	Wa,Wd	Wd = Wa & FF

#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:

```
#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
```

Example Files: None

Also See:	None
------------------	------

#bankx

Syntax:	#BANKX
----------------	---------------

Elements:	None
------------------	------

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

Examples:	<pre>#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.</pre>
------------------	--

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:

```
#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 its 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 `A` 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 */
#build(reset=0x200,interrupt=0x208)

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

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

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

Example None

Files:

Also See: #LOCATE, #RESERVE, #ROM, #ORG

#byte

Syntax: `#byte 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 `int` (8 bit)

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:  #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;

```

Example ex_glint.c

Files:

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:  #case

              int STATUS;

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

```

Example ex_cust.c

Files:

Also See: None

__date__

Syntax: `__DATE__`

Elements: None

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

Examples:

```
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: **#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:

ADC=x	Where x is the number of bits read_adc() should return
ADC=SIGNED	Result returned from read_adc() is signed.(Default is unsigned)
ADC=UNSIGNED	Return result from read_adc() is unsigned.(default is UNSIGNED)
ICD=TRUE	Generates code compatible with Microchips ICD debugging hardware.
ICD=n	For chips with multiple ICSP ports specify the port number being used. The default is 1.
WRITE_EEPROM=ASYNC	Prevents WRITE_EEPROM from hanging while writing is taking place. When used, do not write to EEPROM from both ISR and outside ISR.
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.
HIGH_INTS=TRUE	Use this option for high/low priority interrupts

	on the PIC® 18.
%f=.	No 0 before a decimal point on %f numbers less than 1.
OVERLOAD=KEYWORD	Overloading of functions is now supported. Requires the use of the keyword for overloading.
OVERLOAD=AUTO	Default mode for overloading.
PASS_STRINGS=IN_RAM	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.
CONST=READ_ONLY	Uses the ANSI keyword CONST definition, making CONST variables read only, rather than located in program memory.
CONST=ROM	Uses the CCS compiler traditional keyword CONST definition, making CONST variables located in program memory.
NESTED_INTERRUPTS=TRUE	Enables interrupt nesting for PIC24, dsPIC30, and dsPIC33 devices. Allows higher priority interrupts to interrupt lower priority interrupts.
NORETFIE	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.
NO_DIGITAL_INIT	Normally the compiler sets all I/O pins to digital and turns off the comparator. This option prevents that action.

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. C is case sensitive, all other modes are case insensitive.

CCS2 var16 = NegConst8 is compiled as: var16 = NegConst8 & 0xff (no sign extension)
CCS3 . 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-**

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

Float Options-

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

Compilation mode selection-

```
#device CCS2
```

Example None

Files:

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

Example None

Files:

Also See: #DEVICE

#if expr #else #elif #endif

Syntax:

```
#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:  #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 <i>text</i> #ERROR / warning <i>text</i> #ERROR / information <i>text</i>
Elements:	<i>text</i> 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:	<pre>#if BUFFER_SIZE>16 #error Buffer size is too large #endif #error Macro test: min(x,y)</pre>
Example Files:	ex_psp.c
Also See:	#WARNING

#export (options)

Syntax:	#EXPORT (options)
Elements:	<p><i>FILE=filename</i> 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).</p> <p><i>ONLY=symbol+symbol+.....+symbol</i> 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.</p> <p><i>EXCEPT=symbol+symbol+.....+symbol</i> 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.</p> <p><i>RELOCATABLE</i></p>

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:

```
#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()
*/
```

Example None

Files:

See Also: #IMPORT, #MODULE, Invoking the Command Line Compiler, Multiple Compilation Unit

file

Syntax:	<code>__FILE__</code>
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:	<pre>if(index>MAX_ENTRIES) printf("Too many entries, source file: " "__FILE__ " at line " __LINE__ "\r\n");</pre>
Example Files:	assert.h
Also See:	<code>__line__</code>

filename

Syntax:	<code>__FILENAME__</code>
Elements:	None
Purpose:	The pre-processor identifier is replaced at compile time with the filename of the file being compiled.
Examples:	<pre>if(index>MAX_ENTRIES) printf("Too many entries, source file: " "__FILENAME__ " at line " __LINE__ "\r\n");</pre>
Example Files:	None
Also See:	<code>__line__</code>

#fill_rom

Syntax:	<code>#fill_rom <i>value</i></code>
Elements:	<i>value</i> 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 None

Files:

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 devices .h file in a comment for reference. The PCW device edit utility can modify a particular devices 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 None

Files:

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.

Examples: #HEXCOMMENT Version 3.1 - requires 20MHz crystal

Example None

Files:

Also See: None

#id

Syntax: #ID *number 32*
#ID *number, number, number, number*
#ID "*filename*"
#ID *CHECKSUM*

Elements: *Number 3 2* 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

#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:

```
#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: **#ignore_warnings ALL**
 #IGNORE_WARNINGS NONE
 #IGNORE_WARNINGS *warnings*

Elements: **warnings** is one or more warning numbers separated by commas

Purpose: 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.

Examples:

```
#ignore_warnings 203
while(TRUE) {
#ignore_warnings NONE
```

Example None

Files:

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:

```
#IMPORT(FILE=timer.o, ONLY=TimerTask)
void main(void)
{
    while(TRUE)
        TimerTask();
}
/*
timer.o is linked with this compilation, but only TimerTask()
is visible in scope from this object.
*/
```

Example Files:

None

See Also:

#EXPORT, #MODULE, Invoking the Command Line Compiler, Multiple Compilation Unit

#include

Syntax:

```
#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 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:

```
#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:

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

Example Files: ex_cust.c

Also See: #SEPARATE

#int_xxxx

Syntax:	#INT_AC1	Analog comparator 1 output change
	#INT_AC2	Analog comparator 2 output change
	#INT_AC3	Analog comparator 3 output change
	#INT_AC4	Analog comparator 4 output change
	#INT_ADC1	ADC1 conversion complete
	#INT_ADC2	Analog to digital conversion complete
	#INT_ADCP0	ADC pair 0 conversion complete
	#INT_ADCP1	ADC pair 1 conversion complete
	#INT_ADCP2	ADC pair 2 conversion complete
	#INT_ADCP3	ADC pair 3 conversion complete
	#INT_ADCP4	ADC pair 4 conversion complete
	#INT_ADCP5	ADC pair 5 conversion complete
	#INT_ADDRERR	Address error trap
	#INT_C1RX	ECAN1 Receive Data Ready
	#INT_C1TX	ECAN1 Transmit Data Request
	#INT_C2RX	ECAN2 Receive Data Ready
	#INT_C2TX	ECAN2 Transmit Data Request
	#INT_CAN1	CAN 1 Combined Interrupt Request
	#INT_CAN2	CAN 2 Combined Interrupt Request
	#INT_CNI	Input change notification interrupt
	#INT_COMP	Comparator event
	#INT_CRC	Cyclic redundancy check generator
	#INT_DCI	DCI transfer done
	#INT_DCIE	DCE error
	#INT_DMA0	DMA channel 0 transfer complete
	#INT_DMA1	DMA channel 1 transfer complete
	#INT_DMA2	DMA channel 2 transfer complete
	#INT_DMA3	DMA channel 3 transfer complete
	#INT_DMA4	DMA channel 4 transfer complete
	#INT_DMA5	DMA channel 5 transfer complete
	#INT_DMA6	DMA channel 6 transfer complete
	#INT_DMA7	DMA channel 7 transfer complete
	#INT_DMAERR	DMAC error trap

#INT_EEPROM	Write complete
#INT_EX1	External Interrupt 1
#INT_EX4	External Interrupt 4
#INT_EXT0	External Interrupt 0
#INT_EXT1	External interrupt #1
#INT_EXT2	External interrupt #2
#INT_EXT3	External interrupt #3
#INT_EXT4	External interrupt #4
#INT_FAULTA	PWM Fault A
#INT_FAULTA2	PWM Fault A 2
#INT_FAULTB	PWM Fault B
#INT_IC1	Input Capture #1
#INT_IC2	Input Capture #2
#INT_IC3	Input Capture #3
#INT_IC4	Input Capture #4
#INT_IC5	Input Capture #5
#INT_IC6	Input Capture #6
#INT_IC7	Input Capture #7
#INT_IC8	Input Capture #8
#INT_LOWVOLT	Low voltage detected
#INT_LVD	Low voltage detected
#INT_MATHERR	Arithmetic error trap
#INT_MI2C	Master I2C activity
#INT_MI2C2	Master2 I2C activity
#INT_OC1	Output Compare #1
#INT_OC2	Output Compare #2
#INT_OC3	Output Compare #3
#INT_OC4	Output Compare #4
#INT_OC5	Output Compare #5
#INT_OC6	Output Compare #6
#INT_OC7	Output Compare #7
#INT_OC8	Output Compare #8
#INT_OSC_FAIL	System oscillator failed
#INT_PMP	Parallel master port
#INT_PMP2	Parallel master port 2
#INT_PWM1	PWM generator 1 time based interrupt
#INT_PWM2	PWM generator 2 time based interrupt
#INT_PWM3	PWM generator 3 time based interrupt
#INT_PWM4	PWM generator 4 time based interrupt

#INT_PWMSEM	PWM special event trigger
#INT_QEI	QEI position counter compare
#INT_RDA	RS232 receive data available
#INT_RDA2	RS232 receive data available in buffer 2
#INT_RTC	Real - Time Clock/Calendar
#INT_SI2C	Slave I2C activity
#INT_SI2C2	Slave2 I2C activity
#INT_SPI1	SPI1 Transfer Done
#INT_SPI1E	SPI1E Transfer Done
#INT_SPI2	SPI2 Transfer Done
#INT_SPI2E	SPI2 Error
#INT_SPIE	SPI Error
#INT_STACKERR	Stack Error
#INT_TBE	RS232 transmit buffer empty
#INT_TBE2	RS232 transmit buffer 2 empty
#INT_TIMER1	Timer 1 overflow
#INT_TIMER2	Timer 2 overflow
#INT_TIMER3	Timer 3 overflow
#INT_TIMER4	Timer 4 overflow
#INT_TIMER5	Timer 5 overflow
#INT_TIMER6	Timer 6 overflow
#INT_TIMER7	Timer 7 overflow
#INT_TIMER8	Timer 8 overflow
#INT_TIMER9	Timer 9 overflow
#INT_UART1E	UART1 error
#INT_UART2E	UART2 error
#INT_AUX	Auxiliary memory ISR

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.

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_XXXX HIGHLevel=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: #int_adc
adc_handler() {
    adc_active=FALSE;
}

#int_timer1 noclear
ISR() {
    ...
}

```

Example None

Files:

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

__line__

Syntax: `__line__`

Elements: None

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

Examples:

```
if(index>MAX_ENTRIES)
    printf("Too many entries, source file: "
        "__FILE__" at line " __LINE__ "\r\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 indirected 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:	<p>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.</p> <p>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.</p>
-----------------	---

Examples:	<pre>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. */</pre>
------------------	--

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:	<pre>#NOLIST // Don't clutter up the list file #include <cdriver.h> #LIST</pre>
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:	<pre>#include <18F4520.h> #device ICD=TRUE #OCS 20 MHz #use rs232(debugger) void main() { -----; }</pre>
Example Files:	None
Also See:	#USE DELAY

#opt

Syntax:	#OPT <i>n</i>
Elements:	All Devices: n is the optimization level 0-9
Purpose:	The optimization level is set with this directive. This setting applies to the entire program and may appear anywhere in the file. The default is 9 for normal.
Examples:	<code>#opt 5</code>
Example Files:	None
Also See:	None

#org

Syntax:	#ORG <i>start, end</i> or #ORG <i>segment</i> or #ORG <i>start, end { }</i> or #ORG <i>start, end auto=0</i> #ORG <i>start,end DEFAULT</i> or #ORG <i>DEFAULT</i>
Elements:	<i>start</i> is the first ROM location (word address) to use, <i>end</i> is the last ROM location, <i>segment</i> is the start ROM location from a previous #ORG
Purpose:	<p>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.</p> <p>Follow the ORG with a { } to only reserve the area with nothing inserted by the compiler.</p> <p>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</p>

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).

NULL	NULL
C1OUT	Comparator 1 Output
C2OUT	Comparator 2 Output
C3OUT	Comparator 3 Output
C4OUT	Comparator 4 Output
U1TX	UART1 Transmit
U1RTS	UART1 Request to Send
U2TX	UART2 Transmit
U2RTS	UART2 Request to Send
U3TX	UART3 Transmit
U3RTS	UART3 Request to Send
U4TX	UART4 Transmit
U4RTS	UART4 Request to Send
SDO1	SPI1 Data Output
SCK1OUT	SPI1 Clock Output
SS1OUT	SPI1 Slave Select Output
SDO2	SPI2 Data Output
SCK2OUT	SPI2 Clock Output
SS2OUT	SPI2 Slave Select Output
SDO3	SPI3 Data Output
SCK3OUT	SPI3 Clock Output
SS3OUT	SPI3 Slave Select Output
SDO4	SPI4 Data Output
SCK4OUT	SPI4 Clock Output
SS4OUT	SPI4 Slave Select Output
OC1	Output Compare 1
OC2	Output Compare 2
OC3	Output Compare 3
OC4	Output Compare 4
OC5	Output Compare 5
OC6	Output Compare 6

OC7	Output Compare 7
OC8	Output Compare 8
OC9	Output Compare 9
OC10	Output Compare 10
OC11	Output Compare 11
OC12	Output Compare 12
OC13	Output Compare 13
OC14	Output Compare 14
OC15	Output Compare 15
OC16	Output Compare 16
C1TX	CAN1 Transmit
C2TX	CAN2 Transmit
CSDO	DCI Serial Data Output
CCKOUT	DCI Serial Clock Output
COFSOUT	DCI Frame Sync Output
UPDN1	QE1 Direction Status Output
UPDN2	QE12 Direction Status Output
CTPLS	CTMU Output Pulse
SYNCO1	PWM Synchronization Output Signal
SYNCO2	PWM Secondary Synchronization Output Signal
REFCLKO	REFCLK Output Signal
CMP1	Analog Comparator Output 1
CMP2	Analog Comparator Output 2
CMP3	Analog Comparator Output 3
CMP4	Analog Comparator Output 4
PWM4H	PWM4 High Output
PWM4L	PWM4 Low Output
QE1CCMP	QE1 Counter Comparator Output
QE2CCMP	QE12 Counter Comparator Output
MDOUT	DSM Modulator Output
DCIDO	DCI Serial Data Output
DCISCKOUT	DCI Serial Clock Output
DCIFSOUT	DCI Frame Sync Output
INT1	External Interrupt 1 Input
INT2	External Interrupt 2 Input
INT3	External Interrupt 3 Input
INT4	External Interrupt 4 Input

T1CK	Timer 1 External Clock Input
T2CK	Timer 2 External Clock Input
T3CK	Timer 3 External Clock Input
T4CK	Timer 4 External Clock Input
T5CK	Timer 5 External Clock Input
T6CK	Timer 6 External Clock Input
T7CK	Timer 7 External Clock Input
T8CK	Timer 8 External Clock Input
T9CK	Timer 9 External Clock Input
IC1	Input Capture 1
IC2	Input Capture 2
IC3	Input Capture 3
IC4	Input Capture 4
IC5	Input Capture 5
IC6	Input Capture 6
IC7	Input Capture 7
IC8	Input Capture 8
IC9	Input Capture 9
IC10	Input Capture 10
IC11	Input Capture 11
IC12	Input Capture 12
IC13	Input Capture 13
IC14	Input Capture 14
IC15	Input Capture 15
IC16	Input Capture 16
C1RX	CAN1 Receive
C2RX	CAN2 Receive
OCFA	Output Compare Fault A Input
OCFB	Output Compare Fault B Input
OCFC	Output Compare Fault C Input
U1RX	UART1 Receive
U1CTS	UART1 Clear to Send
U2RX	UART2 Receive
U2CTS	UART2 Clear to Send
U3RX	UART3 Receive
U3CTS	UART3 Clear to Send

U4RX	UART4 Receive
U4CTS	UART4 Clear to Send
SDI1	SPI1 Data Input
SCK1IN	SPI1 Clock Input
SS1IN	SPI1 Slave Select Input
SDI2	SPI2 Data Input
SCK2IN	SPI2 Clock Input
SS2IN	SPI2 Slave Select Input
SDI3	SPI3 Data Input
SCK3IN	SPI3 Clock Input
SS3IN	SPI3 Slave Select Input
SDI4	SPI4 Data Input
SCK4IN	SPI4 Clock Input
SS4IN	SPI4 Slave Select Input
CSDI	DCI Serial Data Input
CCK	DCI Serial Clock Input
COFS	DCI Frame Sync Input
FLTA1	PWM1 Fault Input
FLTA2	PWM2 Fault Input
QEA1	QE1 Phase A Input
QEA2	QE2 Phase A Input
QEB1	QE1 Phase B Input
QEB2	QE2 Phase B Input
INDX1	QE1 Index Input
INDX2	QE2 Index Input
HOME1	QE1 Home Input
HOME2	QE2 Home Input
FLT1	PWM1 Fault Input
FLT2	PWM2 Fault Input
FLT3	PWM3 Fault Input
FLT4	PWM4 Fault Input
FLT5	PWM5 Fault Input
FLT6	PWM6 Fault Input
FLT7	PWM7 Fault Input
FLT8	PWM8 Fault Input
SYNCI1	PWM Synchronization Input 1
SYNCI2	PWM Synchronization Input 2
DCIDI	DCI Serial Data Input
DCISCKIN	DCI Serial Clock Input
DCIFSIN	DCI Frame Sync Input
DTCMP1	PWM Dead Time Compensation 1 Input
DTCMP2	PWM Dead Time Compensation 2 Input
DTCMP3	PWM Dead Time Compensation 3 Input

DTCMP4	PWM Dead Time Compensation 4 Input
DTCMP5	PWM Dead Time Compensation 5 Input
DTCMP6	PWM Dead Time Compensation 6 Input
DTCMP7	PWM Dead Time Compensation 7 Input

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: None

__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 <i>cmd</i>
Elements:	<i>cmd</i> 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.
Examples:	<code>#pragma device PIC16C54</code>
Example Files:	<code>ex_cust.c</code>
Also See:	None

#profile

Syntax:	#profile options												
Elements:	<i>options</i> may be one of the following: <table border="1" data-bbox="346 950 1185 1442"> <tr> <td>function s</td> <td>Profiles the start/end of functions and all profileout() messages.</td> </tr> <tr> <td>func tions, param eters</td> <td>Profiles the start/end of functions, parameters sent to functions, and all profileout() messages.</td> </tr> <tr> <td>profileo ut</td> <td>Only profile profilout() messages.</td> </tr> <tr> <td>paths</td> <td>Profiles every branch in the code.</td> </tr> <tr> <td>off</td> <td>Disable all code profiling.</td> </tr> <tr> <td>on</td> <td>Re-enables the code profiling that was previously disabled with a #profile off command. This will use the last options before disabled with the off command.</td> </tr> </table>	function s	Profiles the start/end of functions and all profileout() messages.	func tions, param eters	Profiles the start/end of functions, parameters sent to functions, and all profileout() messages.	profileo ut	Only profile profilout() messages.	paths	Profiles every branch in the code.	off	Disable all code profiling.	on	Re-enables the code profiling that was previously disabled with a #profile off command. This will use the last options before disabled with the off command.
function s	Profiles the start/end of functions and all profileout() messages.												
func tions, param eters	Profiles the start/end of functions, parameters sent to functions, and all profileout() messages.												
profileo ut	Only profile profilout() messages.												
paths	Profiles every branch in the code.												
off	Disable all code profiling.												
on	Re-enables the code profiling that was previously disabled with a #profile off command. This will use the last options before disabled with the off command.												

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
```

Example Files: `ex_profile.c`

Also See: `#use profile()`, `profileout()`, Code Profile overview

#recursive

Syntax: `#RECURSIVE`

Elements: None

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

Examples:

```
#recursive
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 0x800:0x80B3

Example ex_cust.c

Files:

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 = crc8

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

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

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:

```
#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

Use in place of id parameter, when storing serial number to EEPROM:

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. -or-

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:

```
//Prompt user for serial number to be placed
//at address of serialNumA
//Default serial number = 200int8int8 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: `#task(rate=1s, max=20ms, queue=5)`

Also See: `#USE RTOS`

time

Syntax: `__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:

```
printf("Software was compiled on ");
printf(__TIME__);
```

Example Files: None

Also See: None

#type

Syntax: `#TYPE standard-type=size`
`#TYPE default=area`
`#TYPE unsigned`
`#TYPE signed`
`#TYPE char=signed`
`#TYPE char=unsigned`
`#TYPE ARG=Wx:Wy`
`#TYPE DND=Wx:Wy`
`#TYPE AVOID=Wx:Wy`
`#TYPE RECURSIVE`
`#TYPE CLASSIC`

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 ex_cust.c

Files:

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,0x09,0
//Microsoft
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 use.

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

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 ('_').

Purpose:

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()`.

Examples:

```
#USE
CAPTURE(INPUT=PIN_C2,CAPTURE_RISING,TIMER=
1,FASTEST)
```

Example

None.

Files:

Also See:

`get_capture_time()`, `get_capture_event()`

#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.

Also See: delay_ms(), delay_us()

#use dynamic_memory

Syntax: #USE DYNAMIC_MEMORY

Elements: *None*

Purpose: This pre-processor directive instructs the compiler to create the `_DYNAMIC_HEAD` object. `_DYNAMIC_HEAD` is the location where the first free space is allocated.

Examples: `#USE_DYNAMIC_MEMORY`
`void main () {`
 `}`

Example Files: `ex_malloc.c`

Also See: None

#use fast_io

Syntax: `#USE_FAST_IO (port)`

Elements: *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 (port_outputs=pin, pin?)`

Elements: *port* is A-G, *pin* is one of the pin constants defined in the device's .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.

Examples: `#use fixed_io(a_outputs=PIN_A2, PIN_A3)`

Example Files: None

Also See: `#USE FAST_IO`, `#USE STANDARD_IO`, General Purpose I/O

#use i2c

Syntax: `#USE I2C (options)`

Elements: *Options* are separated by commas and may be:

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

DATA_HOLD	No ACK is sent until I2C_READ is called for data bytes (slave only)
------------------	--

ADDRESS_HOLD	No ACK is sent until I2C_read is called for the address byte (slave only)
---------------------	---

SDA_HOLD	Min of 300ns holdtime on SDA a from SCL goes low
-----------------	--

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:

```
#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:

ICD	Default – configures code profiler to use the ICD connection.
------------	--

TIMER	Optional. If specified, the code profiler run-time o
--------------	--

1	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.
BAUD=x	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.

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:

PWMx or CCPx	Selects the CCP to use, x being the module number to use.
PWMx or OCx	Selects the Output Compare module, x being the module number to use.
OUTPUT=PIN_xx	Selects the PWM pin to use, pin must be one of the CCP OC pins. If device has remappable pins compiler will assign specified pin to specified CCP OC module. If CCP OC module not specified it will assign remappable pin to first available module.
TIMER=x	Selects timer to use with PWM module, default if not specified is timer 2.
FREQUENCY=x	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.

PERIOD=x

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.

BITS=x

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 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:

#use rs232

Syntax: **#USE RS232 (*options*)**
Elements: ***Options*** are separated by commas and may be:

STREAM=id	Associates a stream identifier with this RS232 port. The identifier may then be used in functions like fputc.
BAUD=x	Set baud rate to x
XMIT=pin	Set transmit pin
RCV=pin	Set receive pin
FORCE_SW	Will generate software serial I/O routines even when the UART pins are specified.
BRGH1OK	Allow bad baud rates on chips that have baud rate problems.
ENABLE=pin	The specified pin will be high during transmit. This may be used to enable 485 transmit.
DEBUGGER	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.
RESTART_WDT	Will cause GETC() to clear the WDT as it waits for a character.
INVERT	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.
PARITY=X	Where x is N, E, or O.
BITS =X	Where x is 5-9 (5-7 may not be used with the SCI).
FLOAT_HIGH	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.

ERRORS	Used to cause the compiler to keep receive errors in the variable RS232_ERRORS and to reset errors when they occur.
SAMPLE_EARLY	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.
RETURN=pin	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.
MULTI_MASTER	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.
LONG_DATA	Makes getc() return an int16 and putc accept an int16. This is for 9 bit data formats.
DISABLE_INTS	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.
STOP=X	To set the number of stop bits (default is 1). This works for both UART and non-UART ports.
TIMEOUT=X	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.
SYNC_SLAVE	Makes the RS232 line a synchronous slave, making the receive pin a clock in, and the data pin the data in/out.
SYNC_MASTER	Makes the RS232 line a synchronous master, making the receive pin a clock out, and the data pin

	the data in/out.
SYNC_MATER_CONT	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.
UART1	Sets the XMIT= and RCV= to the chips first hardware UART.
UART1A	Uses alternate UART pins
UART2	Sets the XMIT= and RCV= to the chips second hardware UART.
UART2A	Uses alternate UART pins
NOINIT	No initialization of the UART peripheral is performed. Useful for dynamic control of the UART baudrate or initializing the peripheral manually at a later point in the program's run time. If this option is used, then <code>setup_uart()</code> needs to be used to initialize the peripheral. Using a serial routine (such as <code>getc()</code> or <code>putc()</code>) before the UART is initialized will cause undefined behavior.
ICD	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 <code>#device ICSP=x</code> to specify which pin pair ICD it is connected to. Option is not available when Debugging, see <code>DEBUGGER</code> option above.
UART3	Sets the XMIT= and RCV= to the device's third hardware UART.
UART4	Sets the XMIT= and RCV= to the device's fourth hardware UART.
ICD	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.
Serial Buffer Options:	
RECEIVE_BUFFER=x	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.
TRANSMIT_BUFFER=x	Size in bytes of UART circular transmit buffer,

	default if not specified is zero.
TXISR	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.
NOTXISR	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
Flow Control Options:	
RTS = PIN_xx	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
RTS_LEVEL=x	Specifies the active level of the RTS pin, HIGH is active high and LOW is active low. Defaults to LOW if not specified.
CTS = PIN_xx	Pin to use for CTS flow control. In both FLOW_CONTROL_MODE and SIMPLEX_MODE this pin is sampled to see 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 isn't 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 isn't 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.

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 `ex_cust.c`

Files:

Also See: `getc()`, `putc()`, `printf()`, `setup_uart()`, RS232 I/O overview

#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: options are separated by comma and may be:

timer=X	Where x is 0-4 specifying the timer used by the RTOS.
minor_cycle=time	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 of this time. The compiler can calculate this if it is not specified.
statistics	Maintain min, max, and total time used by each task.

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: *Options* are separated by commas and may be:

MASTER	Set the device as the master. (default)
SLAVE	Set the device as the slave.
BAUD=n	Target bits per second, default is as fast as possible.
CLOCK_HIGH=n	High time of clock in us (not needed if BAUD= is used). (default=0)

CLOCK_LOW=n	Low time of clock in us (not needed if BAUD= is used). (default=0)
DI=pin	Optional pin for incoming data.
DO=pin	Optional pin for outgoing data.
CLK=pin	Clock pin.
MODE=n	The mode to put the SPI bus.
ENABLE=pin	Optional pin to be active during data transfer.
LOAD=pin	Optional pin to be pulsed active after data is transferred.
DIAGNOSTIC=pin	Optional pin to the set high when data is sampled.
SAMPLE_RISE	Sample on rising edge.
SAMPLE_FALL	Sample on falling edge (default).
BITS=n	Max number of bits in a transfer. (default=32)
SAMPLE_COUNT=n	Number of samples to take (uses majority vote). (default=1)
LOAD_ACTIVE=n	Active state for LOAD pin (0, 1).
ENABLE_ACTIVE=n	Active state for ENABLE pin (0, 1). (default=0)
IDLE=n	Inactive state for CLK pin (0, 1). (default=0)
ENABLE_DELAY=n	Time in us to delay after ENABLE is activated. (default=0)
DATA_HOLD=n	Time between data change and clock change
LSB_FIRST	LSB is sent first.
MSB_FIRST	MSB is sent first. (default)
STREAM=id	Specify a stream name for this protocol.
SPI1	Use the hardware pins for SPI Port 1
SPI2	Use the hardware pins for SPI Port 2
FORCE_HW	Use the pic hardware SPI.
SPI3	Use the hardware pins for SPI Port 3
SPI4	Use the hardware pins for SPI Port 4
NOINIT	Don't initialize the hardware SPI Port
XFER16	Uses 16 BIT transfers instead of two 8 BIT transfers

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:

```
#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 None

Files:

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.

Examples:

```
#use standard_io(A)
```

Example ex_cust.c

Files:

Also See: #USE FAST_IO, #USE FIXED_IO, General Purpose I/O

#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()`.

DEFINE=id

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 ('_').

COUNTER or COUNTER=x

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. 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.

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: #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 None

Files:

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 in 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:

```
#USE TOUCHPAD (THRESHOLD=5, PIN_D5='5', PIN_B0='C')
void main(void) {
    char c;
    enable_interrupts(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 Files: `ex_psp.c`

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: `#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 None

Files:

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:

```
#zero_ram
void main() {

}
```

Example Files: ex_cust.c

Also See: None

BUILT-IN FUNCTIONS

BUILT-IN FUNCTIONS

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()	125
sin() cos() tan() asin() acos() atan() sinh() cosh() tanh() atan2()	125
adc_done() adc_done2()	127
assert()	127
atoc	128
atof() atof48() atof64()	129
strtof48()	129
pin_select()	130
atoi() atol() atoi32()	131
atol32() atoi48() atoi64()	131
at_clear_interrupts()	132
at_disable_interrupts()	133
at_enable_interrupts()	133
at_get_capture()	134
at_get_missing_pulse_delay()	135
at_get_period()	135
at_get_phase_counter()	136
at_get_resolution()	137
at_get_set_point()	137
at_get_set_point_error()	138
at_get_status()	138
at_interrupt_active()	139
at_set_compare_time()	140
at_set_missing_pulse_delay()	141
at_set_resolution()	141
at_set_set_point()	142
at_setup_cc()	143
bit_clear()	144
bit_first()	144
bit_last()	145
bit_set()	145
bit_test()	146
bsearch()	147

calloc()	148
ceil()	148
clear_interrupt()	149
cog_status()	149
cog_restart()	150
crc_calc()	150
crc_calc8()	150
crc_calc16()	150
crc_calc32()	150
crc_init(mode)	151
cwg_status()	152
cwg_restart()	153
dac_write()	153
dci_data_received()	154
dci_read()	154
dci_start()	155
dci_transmit_ready()	156
dci_write()	157
delay_cycles()	157
delay_ms()	158
delay_us()	159
disable_interrupts()	160
div()	161
ldiv()	161
dma_start()	162
dma_status()	163
enable_interrupts()	163
erase_program_memory	164
ext_int_edge()	165
fabs()	166
getc() getch() getchar() fgetc()	166
gets() fgets()	167
floor()	168
fmod()	168
printf() fprintf()	169
putc() putchar() fputc()	171
puts() fputs()	172
free()	172
frexp()	173
scanf()	174
get_capture()	176
get_capture()	177
get_capture_ccp1() get_capture_ccp2() get_capture_ccp3() get_capture_ccp4()) get_capture_ccp5()	177
get_capture32_ccp1() get_capture32_ccp2() get_capture32_ccp3() get_capture32_ccp4() get_capture32_ccp5()	179
get_capture_event()	180
get_capture_time()	181

get_capture32()	181
get_hspwm_capture()	182
get_motor_pwm_count()	183
get_nco_accumulator()	183
get_nco_inc_value()	184
get_ticks()	184
get_timerA()	185
get_timerB()	185
get_timerx()	186
get_timerxy()	187
get_timer_ccp1() get_timer_ccp2() get_timer_ccp3() get_timer_ccp4() get_timer_ccp5()	187
get_tris_x()	189
getc() getch() getchar() fgetc()	189
getenv()	190
gets() fgets()	195
goto_address()	196
high_speed_adc_done()	197
i2c_init()	198
i2c_isr_state()	198
i2c_poll()	199
i2c_read()	200
i2c_slaveaddr()	201
i2c_speed()	201
i2c_start()	202
i2c_stop()	203
i2c_write()	203
input()	204
input_change_x()	205
input_state()	206
input_x()	206
interrupt_active()	207
isalnum(char) isalpha(char)	208
iscntrl(x) isdigit(char)	208
isgraph(x) islower(char) isspace(char) isupper(char) isxdigit(char) isprint(x) ispunct(x)	208
isamong()	209
itoa()	210
kbhit()	211
label_address()	212
labs()	212
ldexp()	213
log()	213
log10()	214
longjmp()	215
make8()	215
make16()	216
make32()	216

malloc()	217
memcpy() memmove()	218
memset()	218
modf()	219
_mul()	220
nargs()	220
offsetof() offsetofbit()	221
output_x()	222
output_bit()	223
output_drive()	224
output_float()	224
output_high()	225
output_low()	226
output_toggle()	227
perror()	227
pid_busy()	228
pid_get_result()	228
pid_read()	229
pid_write()	230
pmp_address(address)	231
pmp_output_full() pmp_input_full() pmp_overflow() pmp_error() pmp_timeout()	232
pmp_read()	233
pmp_write()	234
port_x_pullups()	235
pow() pwr()	236
printf() fprintf()	236
profileout()	238
psp_output_full() psp_input_full() psp_overflow()	239
psp_read()	240
psp_write()	241
putc() putchar() fputc()	241
putc_send();	242
fputc_send();	242
puts() fputs()	243
pwm_off()	244
pwm_on()	244
pwm_set_duty()	245
pwm_set_duty_percent	245
pwm_set_frequency	246
qe1_get_count()	247
qe1_set_count()	247
qe1_status()	248
qsort()	248
rand()	249
rcv_buffer_bytes()	250
rcv_buffer_full()	250
read_adc() read_adc2()	251

read_configuration_memory()	252
read_eeprom()	253
read_extended_ram()	253
read_program_memory()	254
read_high_speed_adc()	255
read_rom_memory()	257
read_sd_adc()	257
realloc()	258
release_io()	259
reset_cpu()	259
restart_cause()	260
restart_wdt()	260
rotate_left()	261
rotate_right()	262
rtc_alarm_read()	263
rtc_alarm_write()	263
rtc_read()	264
rtc_write()	265
rtos_await()	265
rtos_disable()	266
rtos_enable()	266
rtos_msg_poll()	267
rtos_msg_read()	267
rtos_msg_send()	268
rtos_overrun()	268
rtos_run()	269
rtos_signal()	270
rtos_stats()	270
rtos_terminate()	271
rtos_wait()	271
rtos_yield()	272
set_adc_channel()	273
set_adc_channel2()	273
set_analog_pins()	273
scanf()	274
set_ccp1_compare_time() set_ccp2_compare_time() set_ccp3_compare_time()	
set_ccp4_compare_time() set_ccp5_compare_time()	277
set_cog_blanking()	278
set_cog_dead_band()	279
set_cog_phase()	280
set_compare_time()	280
set_hspwm_duty()	281
set_hspwm_event()	282
set_hspwm_override()	283
set_hspwm_phase()	283
set_motor_pwm_duty()	284
set_motor_pwm_event()	285
set_motor_unit()	285

set_nco_inc_value()	286
set_pullup()	287
set_pwm1_duty() set_pwm2_duty() set_pwm3_duty() set_pwm4_duty() set_pwm5_duty()	288
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	290
set_ticks()	291
setup_sd_adc_calibration()	291
set_sd_adc_channel()	292
set_timerA()	293
set_timerB()	293
set_timerx()	294
set_timerxy()	294
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	295
set_timer_ccp1() set_timer_ccp2() set_timer_ccp3() set_timer_ccp4() set_timer_ccp5()	296
set_timer_period_ccp1() set_timer_period_ccp2() set_timer_period_ccp3() set_timer_period_ccp4() set_timer_period_ccp5()	297
set_tris_x()	299
set_uart_speed()	299
setjmp()	300
setup_adc(mode)	301
setup_adc2(mode)	301
setup_adc_ports()	302
setup_adc_ports2()	302
setup_adc_reference()	303
setup_at()	303
setup_capture()	304
setup_ccp1() setup_ccp2() setup_ccp3() setup_ccp4() setup_ccp5() setup_ccp6()	305
setup_clc1() setup_clc2() setup_clc3() setup_clc4()	307
setup_comparator()	308
setup_compare()	309
setup_crc(mode)	309
setup_cog()	310
setup_crc()	311
setup_cwg()	312
setup_dac()	313
setup_dci()	314
setup_dma()	315
setup_high_speed_adc()	315
setup_high_speed_adc_pair()	316
setup_hspwm_blanking()	317
setup_hspwm_chop_clock()	318
setup_hspwm_trigger()	319
setup_hspwm_unit()	320
setup_hspwm()	321

<u>setup hspwm unit chop clock()</u>	321
<u>setup low volt detect()</u>	323
<u>setup motor_pwm()</u>	323
<u>setup oscillator()</u>	324
<u>setup pid()</u>	325
<u>setup pmp(option,address_mask)</u>	326
<u>setup power_pwm pins()</u>	327
<u>setup psp(option,address_mask)</u>	328
<u>setup_pwm1()</u> <u>setup_pwm2()</u> <u>setup_pwm3()</u> <u>setup_pwm4()</u>	329
<u>setup_qei()</u>	330
<u>setup_rtc()</u>	331
<u>setup_rtc_alarm()</u>	331
<u>setup_sd_adc()</u>	332
<u>setup_smtx()</u>	333
<u>setup_spi()</u> <u>setup_spi2()</u>	333
<u>setup_timerx()</u>	334
<u>setup_timer_A()</u>	336
<u>setup_timer_B()</u>	336
<u>setup_timer_0()</u>	337
<u>setup_timer_1()</u>	338
<u>setup_timer_2()</u>	338
<u>setup_timer_3()</u>	339
<u>setup_timer_4()</u>	340
<u>setup_timer_5()</u>	341
<u>setup_uart()</u>	341
<u>setup_vref()</u>	342
<u>setup_wdt()</u>	343
<u>setup_zdc()</u>	344
<u>shift_left()</u>	344
<u>shift_right()</u>	345
<u>sleep()</u>	346
<u>smtx_read()</u>	347
<u>smtx_reset_timer()</u>	348
<u>smtx_start()</u>	349
<u>smtx_status()</u>	349
<u>smtx_stop()</u>	350
<u>smtx_write()</u>	350
<u>smtx_update()</u>	351
<u>spi_data_is_in()</u> <u>spi_data_is_in2()</u>	352
<u>spi_init()</u>	352
<u>spi_prewrite(data)</u>	353
<u>spi_read()</u> <u>spi_read2()</u>	353
<u>spi_read3()</u>	353
<u>spi_read4()</u>	353
<u>spi_read_16()</u>	354
<u>spi_read2_16()</u>	354
<u>spi_read3_16()</u>	354
<u>spi_read4_16()</u>	354

<u>spi_speed</u>	355
<u>spi_write()</u> <u>spi_write2()</u>	356
<u>spi_write3()</u>	356
<u>spi_write4()</u>	356
<u>spi_xfer()</u>	357
<u>SPII_XFER_IN()</u>	357
<u>sprintf()</u>	358
<u>sqrt()</u>	359
<u>srand()</u>	359
<u>STANDARD STRING FUNCTIONS()</u> <u>memchr()</u> <u>memcmp()</u> <u>strcat()</u> <u>strchr()</u> <u>strcmp()</u> <u>strcoll()</u> <u>strcspn()</u> <u>strerror()</u> <u>stricmp()</u> <u>strlen()</u> <u>strlwr()</u> <u>strncat()</u> <u>strncmp()</u> <u>strncpy()</u> <u>strpbrk()</u> <u>strrchr()</u> <u>strspn()</u> <u>strstr()</u> <u>strxfrm()</u>	360
<u>strncpy()</u> <u>strncpy()</u>	362
<u>strtod()</u> <u>strtof()</u> <u>strtof48()</u>	362
<u>strtok()</u>	363
<u>strtol()</u>	364
<u>strtoul()</u>	365
<u>swap()</u>	366
<u>tolower()</u> <u>toupper()</u>	366
<u>touchpad_getc()</u>	367
<u>touchpad_hit()</u>	368
<u>touchpad_state()</u>	369
<u>tx_buffer_available()</u>	370
<u>tx_buffer_bytes()</u>	371
<u>tx_buffer_full()</u>	371
<u>va_arg()</u>	372
<u>va_end()</u>	373
<u>va_start</u>	374
<u>write_configuration_memory()</u>	374
<u>write_eeprom()</u>	375
<u>write_extended_ram()</u>	376
<u>write_program_memory()</u>	377
<u>zdc_status()</u>	378

abs()

Syntax: **value = abs(x)**

Parameters: x is any integer or float type.

Returns: Same type as the parameter.

Function: Computes the absolute value of a number.

Availability: All devices

Requires: #INCLUDE <stdlib.h>

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: **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.

sin	returns the sine value of the parameter (measured in radians)
cos	returns the cosine value of the parameter (measured in radians)
tan	returns the tangent value of the parameter (measured in radians)
asin	returns the arc sine value in the range $[-\pi/2, +\pi/2]$ radians
acos	returns the arc cosine value in the range $[0, \pi]$ radians
atan	returns the arc tangent value in the range $[-\pi/2, +\pi/2]$ radians
atan2	returns the arc tangent of y/x in the range $[-\pi, +\pi]$ radians
sinh	returns the hyperbolic sine of x
cosh	returns the hyperbolic cosine of x
tanh	returns the hyperbolic tangent of x

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:

```
float phase;
// Output one sine wave
for(phase=0; phase<2*3.141596; phase+=0.01)
    set_analog_voltage( sin(phase)+1 );
```

Example Files: ex_tank.c

Also See: log(), log10(), exp(), pow(), sqrt()

adc_done() adc_done2()

Syntax: **value = adc_done();**
 value = adc_done2();

Parameters: None

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:

```
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_START_ONLY);

int1 done = adc_done();
while(!done) {
    done = adc_done();
}
value = read_adc(ADC_READ_ONLY);
printf("A/C value = %LX\n\r", value);
}
```

Example None

Files:

Also See: setup_adc(), set_adc_channel(), setup_adc_ports(), read_adc(),
 ADC Overview

assert()

Syntax: **assert (condition);**

Parameters: **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:

```
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
```

Example Files: None

Also See: #USE RS232, RS232 I/O Overview

atof

Syntax: **atof(string);**

Parameters: *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:

```
char string [10];
float32 x;

strcpy (string, "12E3");
x = atof(string);
// x is now 12000.00
```

Example Files:	None
Also See:	atoi(), atol(), atoi32(), atof(), printf()

atof() atof48() atof64() strtof48()

Syntax: **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:

```
char string [10];
float x;

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 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: **ival** = **atoi(string)**
 or
 lvalue = **atol(string)**
 or
 i32value = **atoi32(string)**
 or
 i48value=**atoi48(string)**
 or
 i64value=**atoi64(string)**
 or
 L32vale=**atol32(string)**

Parameters: **string** is a pointer to a null terminated string of characters.

Returns: **ival** 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.

Availability: All devices

Requires: #INCLUDE <stdlib.h>

Examples:

```
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: **at_clear_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 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 None

Files:

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.

Availability: All devices with an AT module.

Requires: Constants defined in the device's header file

Examples: `at_disable_interrupts(AT_PHASE_INTERRUPT);`
`at_disable_interrupts(AT_PERIOD_INTERRUPT|AT_CC1_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_clear_interrupts()`, `at_interrupt_active()`, `at_setup_cc()`, `at_set_compare_time()`, `at_get_capture()`, `at_get_status()`, `setup_at()`

at_enable_interrupts()

Syntax: `at_enable_interrupts(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_interrupts(AT_PHASE_INTERRUPT);
at_enable_interrupts(AT_PERIOD_INTERRUPT|AT_CC1_INTERRUPT);
```

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_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_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_status()

at_get_missing_pulse_delay()

Syntax: **result=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:

```
result=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_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_period()

Syntax: **result=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_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_get_period()`, `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 None

Files:

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:	<code>at_set_resolution()</code> , <code>at_get_resolution()</code> , <code>at_set_missing_pulse_delay()</code> , <code>at_get_missing_pulse_delay()</code> , <code>at_get_period()</code> , <code>at_get_phase_counter()</code> , <code>at_set_set_point()</code> , <code>at_get_set_point_error()</code> , <code>at_enable_interrupts()</code> , <code>at_disable_interrupts()</code> , <code>at_clear_interrupts()</code> , <code>at_interrupt_active()</code> , <code>at_setup_cc()</code> , <code>at_set_compare_time()</code> , <code>at_get_capture()</code> , <code>at_get_status()</code> , <code>setup_at()</code>

at_get_set_point_error()

Syntax: `result=at_get_set_point_error();`

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: `result=at_get_set_point_error();`

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_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_THEN_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_PERIOD_INTERRUPT
- AT_CC3_INTERRUPT
- AT_CC2_INTERRUPT
- AT_CC1_INTERRUPT

Returns: TRUE if the specified AT interrupt's flag is set, interrupt is active, or FALSE if the flag is clear, interrupt is not active.

Function: To check if the specified Angular Timer interrupt flag is set.

Availability: All devices with an AT module.

Requires: Constants defined in the device's header file

Examples:

```
#INT-AT1
voidl_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_clear_interrupts()`, `at_setup_cc()`, `at_set_compare_time()`, `at_get_capture()`, `at_get_status()`, `setup_at()`

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);`

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_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
- 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:

```
at_setup_cc(1, AT_CC_ENABLED|AT_CC_CAPTURE_MODE |
AT_CAPTURE_FALLING_EDGE|AT_CAPTURE_INPUT_ATCAP);

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_get_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 a 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:

```
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:

```
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:

```
int x;  
x=5;  
bit_set(x,3);  
// x is now 13
```

Example Files: ex_patg.c

Also See: bit_clear(), bit_test()

bit_test()

Syntax: `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:

```
if( bit_test(x,3) || !bit_test (x,1) ){  
    //either bit 3 is 1 or bit 1 is 0  
}  
  
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
```

Example ex_patg.c

Files:**Also See:** bit_clear(), bit_set()

bsearch()

Syntax: ip = bsearch (&key, base, num, width, compare)**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:**

```
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);
}

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: `ptr=calloc(nmem, size)`

Parameters: *nmem* is an integer representing the number of member objects
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 *size*. The space is initialized to all bits zero.

Availability: All devices

Requires: `#INCLUDE <stdlibm.h>`

Examples:

```
int * iptr;
iptr=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: `result = ceil (value)`

Parameters: *value* is any float type

Returns: A float with precision equal to *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()

cog_status()

Syntax: value=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:

```
if (cog_status() == COG_AUTO_SHUTDOWN)
    cog_restart();
```

Example Files: None

Also See: `setup_cog()`, `set_cog_dead_band()`,
`set_cog_blanking()`, `set_cog_phase()`,
`cog_restart()`

cog_restart()

Syntax: `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:

```
if(cog_status()==COG_AUTO_SHUTDOWN)
    cog_restart();
```

Example Files: None

Also See: `setup_cog()`, `set_cog_dead_band()`,
`set_cog_blanking()`, `set_cog_phase()`,
`cog_status()`

crc_calc()

crc_calc8()

crc_calc16()

crc_calc32()

Syntax: `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 value of the function, 8-bit for `crc_calc8()`, 16-bit for `crc_calc16()` and 32-bit for `crc_calc32()`. 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:

```

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:	<pre> crc_init (); // Starts the CRC accumulator out at 0 crc_init(0xFEEE); // Starts the CRC accumulator out at 0xFEEE </pre>
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:	<pre> if(cwg_status() == CWG_AUTO_SHUTDOWN) cwg_restart(); </pre>
Example Files:	None
Also See:	setup_cwg(), cwg_restart()

cwg_restart()

Syntax:	<code>cwg_restart();</code>
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:	<pre>if(cwg_status() == CWG_AUTO_SHUTDOWN) cwg_restart();</pre>
Example Files:	None
Also See:	<code>setup_cwg()</code> , <code>cwg_status()</code>

dac_write()

Syntax:	<code>dac_write (value)</code> <code>dac_write (channel, value)</code>
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 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:	<pre>int i = 0; setup_dac(DAC_VDD DAC_OUTPUT); while(1) {</pre>

```

        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

dc_i_data_received()

Syntax: dc_i_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()

dc_i_read()

Syntax: dc_i_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);
}
```

Example Files: None

Also See: `DCI Overview`, `setup_dci()`, `dci_start()`, `dci_write()`, `dci_transmit_ready()`, `dci_data_received()`

dci_start()

Syntax: `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: `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_write()`, `dci_read()`, `dci_transmit_ready()`, `dci_data_received()`

dc_i_transmit_ready()

Syntax: `dc_i_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:

```
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()`

dc_i_write()

Syntax: `dc_i_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_dc_i()`), 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:

```
while(1)
{
    dc_i_read(&left_channel, &right_channel);
    dc_i_write(&left_channel, &right_channel);
}
```

Example Files: None

Also See: DCI Overview, `setup_dc_i()`, `dc_i_start()`, `dc_i_read()`, `dc_i_transmit_ready()`, `dc_i_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:

```
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:

```
#use delay (clock=20000000)

delay_ms( 2 );

void delay_seconds(int n) {
    for (;n!=0; n- -)
        delay_ms( 1000 );
}
```

Example Files: ex_sqw.c

Also See: delay_us(), delay_cycles(), #USE_DELAY

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:

```
#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()

div()

ldiv()

Syntax: **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:

```

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

```

Example Files: None

Also See: None

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_CONTINUOUS | 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: `Value = dma_status(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.

INTR_GLOBAL – Enables all interrupt levels (same as INTR_LEVEL0)

INTR_NORMAL – Use normal vectors for the ISR

INTR_ALTERNATE – Use alternate vectors for the ISR

INTR_LEVEL0 .. INTR_LEVEL7 – Enables interrupts at this level and above, interrupts at lower levels are disabled

INTR_CN_PIN | PIN_xx – Enables a CN pin interrupts

Availability: All dsPIC and PIC24 devices

Requires: Should have a #INT_xxxx, Constants are defined in the devices .h file.

Examples:

```
enable_interrupts(INT_TIMER0);
enable_interrupts(INT_TIMER1);
enable_interrupts(INTR_CN_PIN|Pin_B0);
```

Example Files: None

Also See: disable_enterrupts(), #INT_xxxx, Interrupts Overview, clear_interrupt()
interrupt_active()

erase_program_memory

Syntax: `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. FLASH_ERASE_SIZE varies depending on the part.

Family	FLASH_ERASE_SIZE
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

ext_int_edge()

Syntax: `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:

```
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_wakeup.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:	<pre>double result; result=fabs(-40.0) // result is 40.0</pre>
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:	<p>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®.</p> <p>If fgetc() is used then the specified stream is used where getc() defaults to STDIN</p>

(the last USE RS232).

Availability: All devices

Requires: #USE RS232

Examples:

```
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 ex_stwt.c

Files:

Also See: putc(), kbhit(), printf(), #USE RS232, input.c, RS232 I/O Overview

gets() fgets()

Syntax: **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 getch()) 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:	<pre>char string[30]; printf("Password: "); gets(string); if(strcmp(string, password)) printf("OK");</pre>
Example Files:	None
Also See:	getc(), get_string in input.c

floor()

Syntax:	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:	<pre>// Find the fractional part of a value frac = value - floor(value);</pre>
Example Files:	None
Also See:	ceil()

fmod()

Syntax:	result= fmod (val1, val2)
----------------	----------------------------------

Parameters:	val1 is any float type val2 is any float type
Returns:	result is a float with precision equal to input parameters val1 and val2
Function:	Returns the floating point remainder of val1/val2. Returns the value val1 - i*val2 for some integer "i" such that, if val2 is nonzero, the result has the same sign as val1 and magnitude less than the magnitude of val2.
Availability:	All devices.
Requires:	#INCLUDE <math.h>
Examples:	<pre>float result; result=fmod(3,2); // result is 1</pre>
Example Files:	None
Also See:	None

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. 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). Note that format specifics do not work in ram band strings.
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:

c	Character
s	String or character
u	Unsigned int
d	Signed int
Lu	Long unsigned int
Ld	Long signed int
x	Hex int (lower case)
X	Hex int (upper case)
Lx	Hex long int (lower case)
LX	Hex long int (upper case)
f	Float with truncated decimal
g	Float with rounded decimal
e	Float in exponential format
w	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.

Example formats:

Specifier	Value=0x12	Value=0xfe
%03u	018	254
%u	18	254
%2u	18	*
%5	18	254
%d	18	-2
%x	12	fe
%X	12	FE
%4X	0012	00FE
%3.1w	1.8	25.4

* Result is undefined - Assume garbage.

Availability: All Devices

Requires: #USE RS232 (unless fname is used)

Examples:

```
byte x,y,z;
printf("HiThere");
printf("RTCCValue=>%2x\n\r",get_rtcc());
printf("%2u %X %4X\n\r",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 STDOUT (the last USE RS232).

Availability: All devices

Requires: #USE RS232

Examples:

```
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:	puts(" ----- "); puts(" HI "); puts(" ----- ");
Example Files:	None
Also See:	printf(), gets(), RS232 I/O Overview

free()

Syntax:	free(ptr)
Parameters:	ptr 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 <stdlibm.h>

Examples:

```
int * iptr;
iptr=malloc(10);
free(iptr)
// iptr will be deallocated
```

Example Files: None

Also See: realloc(), malloc(), calloc()

frexp()

Syntax: **result=frexp (value, &exp);**

Parameters: **value** is any float type
exp is a signed int.

Returns: result is a float with precision equal to **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 to 1) or zero, such that value is result times 2 raised to power exp. If value is zero then both parts are zero.

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 %*n**t*. *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:

- c** 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.
- s** 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.
- u** Matches an unsigned decimal integer. The corresponding argument shall be a pointer to an unsigned integer.
- Lu** Matches a long unsigned decimal integer. The corresponding argument shall be a pointer to a long unsigned integer.
- d** Matches a signed decimal integer. The corresponding argument shall be a pointer to a signed integer.

Ld	Matches a long signed decimal integer. The corresponding argument shall be a pointer to a long signed integer.
o	Matches a signed or unsigned octal integer. The corresponding argument shall be a pointer to a signed or unsigned integer.
Lo	Matches a long signed or unsigned octal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.
x or X	Matches a hexadecimal integer. The corresponding argument shall be a pointer to a signed or unsigned integer.
Lx or LX	Matches a long hexadecimal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.
i	Matches a signed or unsigned integer. The corresponding argument shall be a pointer to a signed or unsigned integer.
Li	Matches a long signed or unsigned integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.
f,g or e	Matches a floating point number in decimal or exponential format. The corresponding argument shall be a pointer to a float.
[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, <code>%[a-z]</code> would include all characters from a to z in the set and <code>%[^a-z]</code> 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.
n	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. 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]`, `[%c]` or `[%n]`, unless a white-space character precedes the `[%c]` or `[%c]` specifiers.

Availability: All Devices

Requires: #USE RS232

Examples:

```
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_ccmp.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 None

Files:

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()`

`get_capture32_ccp1()`
`get_capture32_ccp2()`
`get_capture32_ccp3()`
`get_capture32_ccp4()`
`get_capture32_ccp5()`

Syntax: `value=get_capture32_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: `value32` -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, 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.

Availability: Available only on PIC24FxxKMxxx family of devices with a MCCP and/or SCCP modules.

Requires: Nothing

Examples:

```
unsigned int32 value;

setup_ccp1(CCP_CAPTURE_FE|CCP_TIMER_32_BIT);

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_capture_ccpX()

get_capture_event()

Syntax: 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:

```
#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: `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:

```
#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: `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:

```
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: Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

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

Examples: `Data16 = get_motor_pwm_count(1);`

Example Files: None

Also See: `setup_motor_pwm()`, `set_motor_unit()`, `set_motor_pwm_event()`,
`set_motor_pwm_duty()`;

get_nco_accumulator()

Syntax: `value =get_nco_accumulator();`

Parameters: none

Returns: current value of accumulator.

Availability: On devices with a NCO module.

Examples: `value = get_nco_accumulator();`

Example Files: None

Also See: `setup_nco()`, `set_nco_inc_value()`,
`get_nco_inc_value()`

get_nco_inc_value()

Syntax:	<code>value =get_nco_inc_value();</code>
Parameters:	None
Returns:	- current value set in increment registers.
Availability:	On devices with a NCO module.
Examples:	<code>value = get_nco_inc_value();</code>
Example Files:	None
Also See:	<code>setup_nco(), set_nco_inc_value(), get_nco_accumulator()</code>

get_ticks()

Syntax:	<code>value = get_ticks([stream]);</code>
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:	<pre>#USE TIMER(TIMER=1,TICK=1ms,BITS=16,NOISR) void main(void) { unsigned int16 current_tick; current_tick = get_ticks(); }</pre>
Example Files:	None

Also See: **#USE TIMER, set_ticks()**

get_timerA()

Syntax: **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:

```
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()  
value=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:

```
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: **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:

```
if(get_timer23() > TRIGGER_TIME)
    ExecuteEvent();
```

Example Files: ex_stwt.c

Also See: Timer Overview, setup_timerX(), get_timerXY(), set_timerX(), set_timerXY()

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                                  //lower timer
value16=get_timer_ccpx(0);          //get the 16 bit timer
value from

value16=get_timer_ccpx(1);          //get the 16 bit timer
value from

//upper timer
```

Example Files: None

Also See: set_pwmX_duty(), setup_ccpX(), set_ccpX_compare_time(), set_timer_ccpX(), set_timer_period_ccpX(), get_capture_ccpX(), get_captures32_ccpX()

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()`

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:

```
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

getenv()

Syntax: `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.

FUSE_SET:ffff	Returns 1 if fuse ffff is enabled
FUSE_VALID:ffff	Returns 1 if fuse ffff is valid
INT:iiii	Returns 1 if the interrupt iiii is valid
ID	Returns the device ID (set by #ID)
DEVICE	Returns the device name string (like "PIC16C74")
CLOCK	Returns the MPU FOSC
VERSION	Returns the compiler version as a float
VERSION_STRING	Returns the compiler version as a string
PROGRAM_MEMORY	Returns the size of memory for code (in words)
STACK	Returns the stack size
SCRATCH	Returns the start of the compiler scratch area
DATA_EEPROM	Returns the number of bytes of data EEPROM
EEPROM_ADDRESS	Returns the address of the start of EEPROM. 0 if not supported by the device.
READ_PROGRAM	Returns a 1 if the code memory can be read
ADC_CHANNELS	Returns the number of A/D channels
ADC_RESOLUTION	Returns the number of bits returned from READ_ADC()
ICD	Returns a 1 if this is being compiled for a ICD
SPI	Returns a 1 if the device has SPI

USB	Returns a 1 if the device has USB
CAN	Returns a 1 if the device has CAN
I2C_SLAVE	Returns a 1 if the device has I2C slave H/W
I2C_MASTER	Returns a 1 if the device has I2C master H/W
PSP	Returns a 1 if the device has PSP
COMP	Returns a 1 if the device has a comparator
VREF	Returns a 1 if the device has a voltage reference
LCD	Returns a 1 if the device has direct LCD H/W
UART	Returns the number of H/W UARTs
AUART	Returns 1 if the device has an ADV UART
CCPx	Returns a 1 if the device has CCP number x
TIMERx	Returns a 1 if the device has TIMER number x
FLASH_WRITE_SIZE	Smallest number of bytes that can be written to FLASH
FLASH_ERASE_SIZE	Smallest number of bytes that can be erased in FLASH
BYTES_PER_ADDRESS	Returns the number of bytes at an address location
BITS_PER_INSTRUCTION	Returns the size of an instruction in bits
RAM	Returns the number of RAM bytes available for your device.

Built-in Functions

SFR:name	Returns the address of the specified special file register. The output format can be used with the preprocessor command #bit. name must match SFR denomination of your target PIC (example: STATUS, INTCON, TXREG, RCREG, etc)
BIT:name	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 #byte. name must match SFR.bit denomination of your target PIC (example: C, Z, GIE, TMR0IF, etc)
SFR_VALID:name	Returns TRUE if the specified special file register name is valid and exists for your target PIC (example: getenv("SFR_VALID:INTCON"))
BIT_VALID:name	Returns TRUE if the specified special file register bit is valid and exists for your target PIC (example: getenv("BIT_VALID:TMR0IF"))
PIN:PB	Returns 1 if PB is a valid I/O PIN (like A2)
UARTx_RX	Returns UARTxPin (like PINxC7)
UARTx_TX	Returns UARTxPin (like PINxC6)
SPIx_DI	Returns SPIxDI Pin
SPIxDO	Returns SPIxDO Pin
SPIxCLK	Returns SPIxCLK Pin
ETHERNET	Returns 1 if device supports Ethernet
QEI	Returns 1 if device has QEI

DAC	Returns 1 if device has a D/A Converter
DSP	Returns 1 if device supports DSP instructions
DCI	Returns 1 if device has a DCI module
DMA	Returns 1 if device supports DMA
CRC	Returns 1 if device has a CRC module
CWG	Returns 1 if device has a CWG module
NCO	Returns 1 if device has a NCO module
CLC	Returns 1 if device has a CLC module
DSM	Returns 1 if device has a DSM module
OPAMP	Returns 1 if device has op amps
RTC	Returns 1 if device has a Real Time Clock
CAP_SENSE	Returns 1 if device has a CSM cap sense module and 2 if it has a CTMU module
EXTERNAL_MEMORY	Returns 1 if device supports external program memory
INSTRUCTION_CLOCK	Returns the MPU instruction clock
ENH16	Returns 1 for Enhanced 16 devices
ENH24	Returns 2 for Enhanced 24 devices
IC	Returns number of Input Capture units device has
ICx	Returns TRUE if ICx is on this part
OC	Returns number of Output Compare units device has

OCx	Returns TRUE if OCx is on this part
RAM_START	Returns the starting address of the first general purpose RAM location
PSV	Returns TRUE if program space visibility (PSV) is enabled. If PSV is enabled, data in program memory ('const char *' or 'rom char *') can be assigned to a regular RAM pointer ('char *') and a regular RAM pointer can dereference data from program memory or RAM.

Availability: All devices

Requires: Nothing

Examples:

```
#IF getenv("VERSION")<3.050
    #ERROR Compiler version too old
#endif

for(i=0;i<getenv("DATA_EEPROM");i++)
    write_eeprom(i,0);

#ifdef FUSE_VALID:BROWNOUT
    #FUSE BROWNOUT
#endif

#byte status_reg=GETENV("SFR:STATUS")

#bit carry_flag=GETENV("BIT:C")
```

Example Files: None

Also See: None

gets() fgets()

Syntax: **gets(*string*)**
value = fgets(*string*, *stream*)

Parameters:	<i>string</i> is a pointer to an array of characters. <i>Stream</i> is a stream identifier (a constant byte)
Returns:	undefined
Function:	Reads characters (using <code>getc()</code>) into the string until a RETURN (value 13) is encountered. The string is terminated with a 0. Note that <code>INPUT.C</code> has a more versatile <code>get_string</code> function. If <code>fgets()</code> is used then the specified stream is used where <code>gets()</code> defaults to <code>STDIN</code> (the last USE RS232).
Availability:	All devices
Requires:	#USE RS232
Examples:	<pre>char string[30]; printf("Password: "); gets(string); if(strcmp(string, password)) printf("OK");</pre>
Example Files:	None
Also See:	<code>getc()</code> , <code>get_string</code> in <code>input.c</code>

goto_address()

Syntax:	<code>goto_address(location);</code>
Parameters:	<code>location</code> is a ROM address, 16 or 32 bit int.
Returns:	Nothing
Function:	This function jumps to the address specified by <code>location</code> . 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:	<pre>#define LOAD_REQUEST PIN_B1 #define LOADER 0x1f00</pre>


```
if(input(Load_Request))
    goto_address(Loader);
```

Example Files: setjmp.h

Also See: label_address()

high_speed_adc_done()

Syntax: 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.

Availability: Only on dsPIC33FJxxGSxxx devices.

Requires: None

Examples:

```
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\r", 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: `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: #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

Example Files: None

Also See: I2C_POLL(), i2c_speed(), I2C_SlaveAddr(), I2C_ISR_STATE(_), I2C_WRITE(), I2C_READ(), _USE_I2C(), I2C()

i2c_isr_state()

Syntax: `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:

```
#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)
        rcv_buffer[state - 1] = i2c_read();
}
```

Example Files: `ex_slave.c`

Also See: `i2c_poll`, `i2c_speed`, `i2c_start`, `i2c_stop`, `i2c_slaveaddr`, `i2c_write`, `i2c_read`, `#USE I2C`, `I2C Overview`

`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
```

Example Files: None

Also See: i2c_speed, i2c_start, i2c_stop, i2c_slaveaddr, i2c_isr_state, i2c_write, i2c_read, #USE I2C, I2C Overview

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:

```
i2c_start();
i2c_write(0xa1);
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: **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.

Availability: Devices with built in I2C

Requires: #USE I2C

Examples:

```
i2c_SlaveAddr(0x08);
i2c_SlaveAddr(i2cStream1, 0x08);
```

Example Files: ex_slave.c

Also See: i2c_poll, i2c_speed, i2c_start, i2c_stop, i2c_isr_state, i2c_write, i2c_read, #USE I2C, I2C Overview

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:	<code>I2C_Speed (400000);</code>
Example Files:	none
Also See:	<code>i2c_poll</code> , <code>i2c_start</code> , <code>i2c_stop</code> , <code>i2c_slaveaddr</code> , <code>i2c_isr_state</code> , <code>i2c_write</code> , <code>i2c_read</code> , #USE I2C, I2C Overview

i2c_start()

Syntax:	<code>i2c_start()</code> <code>i2c_start(stream)</code> <code>i2c_start(stream, restart)</code>
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:	<pre> 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(); </pre>
------------------	--

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
------------------	--

i2c_stop()

Syntax:	i2c_stop() i2c_stop(stream)
----------------	--

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:	<pre> i2c_start(); // Start condition i2c_write(0xa0); // Device address i2c_write(5); // Device command i2c_write(12); // Device data i2c_stop(); // Stop condition </pre>
------------------	---

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:	i2c_write (data) i2c_write (stream, data)
----------------	--

Parameters:	<i>data</i> is an 8 bit int <i>stream</i> - specify the stream defined in #USE I2C
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:	<pre>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</pre>
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:	value = input (<i>pin</i>)
Parameters:	<i>Pin</i> 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 . 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.
Requires:	Pin constants are defined in the devices .h file
Examples:	<pre>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</pre>
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: `value = input_x(a)`

```

value = input_b()
value = input_c()
value = input_d()
value = input_e()
value = input_f()
value = input_g()
value = input_h()
value = input_j()
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: `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: `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 `#INT_xxxx`, Constants are defined in the devices .h file.

Examples: `interrupt_active(INT_TIMER0);`
`interrupt_active(INT_TIMER1);`

Example Files:	None
-----------------------	------

Also See:	<code>disable_interrupts()</code> , <code>#INT</code> , Interrupts Overview <code>clear_interrupt</code> , <code>enable_interrupts()</code>
------------------	--

isalnum(char)

isalpha(char)

isctrl(x)

isdigit(char)

isgraph(x)

islower(char)

isspace(char)

isupper(char)

isxdigit(char)

isprint(x)

ispunct(x)

Syntax:	<code>value = isalnum(<i>datac</i>)</code> <code>value = isalpha(<i>datac</i>)</code> <code>value = isdigit(<i>datac</i>)</code> <code>value = islower(<i>datac</i>)</code> <code>value = isspace(<i>datac</i>)</code> <code>value = isupper(<i>datac</i>)</code> <code>value = isxdigit(<i>datac</i>)</code> <code>value = isctrl(<i>datac</i>)</code> <code>value = isgraph(<i>datac</i>)</code> <code>value = isprint(<i>datac</i>)</code> <code>value = punct(<i>datac</i>)</code>
----------------	--

Parameters:	<i>datac</i> is a 8 bit character
--------------------	-----------------------------------

Returns:	0 (or FALSE) if <i>datac</i> dose not match the criteria, 1 (or TRUE) if <i>datac</i> does match the criteria.
-----------------	--

Function:	Tests a character to see if it meets specific criteria as follows: <u>isalnum(x) X is 0..9, 'A'..'Z', or 'a'..'z'</u>
------------------	--

isalpha(x)	X is 'A'..'Z' or 'a'..'z'
isdigit(x)	X is '0'..'9'
islower(x)	X is 'a'..'z'
isupper(x)	X is 'A'..'Z'
isspace(x)	X is a space
isxdigit(x)	X is '0'..'9', 'A'..'F', or 'a'..'f'
iscntrl(x)	X is less than a space
isgraph(x)	X is greater than a space
isprint(x)	X is greater than or equal to a space
ispunct(x)	X is greater than a space and not a letter or number

Availability: All devices.

Requires: #INCLUDE <ctype.h>

Examples:

```
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 string

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: `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:

```
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;
        return(0);
    }
}
```

Example Files: ex_tgetc.c

Also See: getc(), #USE RS232, RS232 I/O Overview

label_address()

Syntax: `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:

```
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: `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:

```
if(labs( target_value - actual_value ) > 500)
    printf("Error is over 500 points\r\n");
```

Example Files: None

Also See: abs()

ldexp()

Syntax:	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:	<pre>float result; result=ldexp(.5,0); // result is .5</pre>
Example Files:	None
Also See:	frexp(), exp(), log(), log10(), modf()

log()

Syntax:	result = log (value)
Parameters:	value is any float type
Returns:	A float with precision equal to value
Function:	<p>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.</p> <p>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.</p> <p>Domain error occurs in the following cases:</p>

	<ul style="list-style-type: none">• <code>log</code>: when the argument is negative
Availability:	All devices
Requires:	<code>#INCLUDE <math.h></code>
Examples:	<code>lnx = log(x);</code>
Example Files:	None
Also See:	<code>log10()</code> , <code>exp()</code> , <code>pow()</code>

log10()

Syntax:	result = log10 (value)
Parameters:	value is any float type
Returns:	A float with precision equal to value
Function:	<p>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.</p> <p>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.</p> <p>Domain error occurs in the following cases:</p> <ul style="list-style-type: none">• <code>log10</code>: when the argument is negative
Availability:	All devices
Requires:	<code>#INCLUDE <math.h></code>
Examples:	<code>db = log10(read_adc()*(5.0/255))*10;</code>
Example Files:	None
Also See:	<code>log()</code> , <code>exp()</code> , <code>pow()</code>

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 = MAKE8(var, offset)`

Parameters: *var* is a 16 or 32 bit integer.
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 = (((var >> (offset*8)) & 0xff)` except it is done with a single byte move.

Availability: All devices

Requires: Nothing

Examples:

```
int32 x;
int y;

y = make8(x,3); // Gets MSB of x
```

Example Files:	None
Also See:	make16(), make32()

make16()

Syntax:	i16 = MAKE16(<i>varhigh</i>, <i>varlow</i>)
Parameters:	<i>varhigh</i> and <i>varlow</i> 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: <code>i16 = (int16)(varhigh&0xff)*0x100+(varlow&0xff)</code> except it is done with two byte moves.
Availability:	All devices
Requires:	Nothing
Examples:	<pre>long x; int hi,lo; x = make16(hi,lo);</pre>
Example Files:	ltc1298.c
Also See:	make8(), make32()

make32()

Syntax:	i32 = MAKE32(<i>var1</i>, <i>var2</i>, <i>var3</i>, <i>var4</i>)
Parameters:	<i>var1-4</i> are a 8 or 16 bit integers. <i>var2-4</i> 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:	<pre>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</pre>
Example Files:	ex_freq.c
Also See:	make8(), make16()

malloc()

Syntax:	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:	<pre>int * iptr; iptr=malloc(10); // iptr will point to a block of memory of 10 bytes.</pre>
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);

char a[20]="hello";
memmove(a, a+2, 5);
// a is now "llo"
```

Example Files: None

Also See: strcpy(), memset()

memset()

Syntax: `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:	<pre>memset(arrayA, 0, sizeof(arrayA)); memset(arrayB, '?', sizeof(arrayB)); memset(&structA, 0xFF, sizeof(structA));</pre>
Example Files:	None
Also See:	memcpy()

modf()

Syntax:	result= modf (value, & 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:	<pre>float 48 result, integral; result=modf(123.987,&integral); // result is .987 and integral is 123.0000</pre>
Example Files:	None
Also See:	None

`_mul()`

Syntax: `prod=_mul(val1, val2);`

Parameters: `val1` and `val2` are both 8-bit, 16-bit, or 48-bit integers

Returns:

<i>val1</i>	<i>val2</i>	<i>prod</i>
8	8	16
16*	16	32
32*	32	64
48*	48	64**

* 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:

```
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 than 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:

```
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: `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:

```
struct time_structure {
    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:	<code>OUTPUT_B(0xf0);</code>
Example Files:	<code>ex_patg.c</code>
Also See:	<code>input()</code> , <code>output_low()</code> , <code>output_high()</code> , <code>output_float()</code> , <code>output_bit()</code> , <code>#USE FIXED_IO</code> , <code>#USE FAST_IO</code> , <code>#USE STANDARD_IO</code> , General Purpose I/O

output_bit()

Syntax:	<code>output_bit(<i>pin</i>, <i>value</i>)</code>
Parameters:	<p>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 <code>0x2C2*8+3</code> or 5651 . This is defined as follows: <code>#define PIN_A3 5651</code> . The PIN could also be a variable. The variable must have a value equal to one of the constants (like <code>PIN_A1</code>) to work properly. The tristate register is updated unless the <code>FAST_IO</code> mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time.</p> <p>Value is a 1 or a 0.</p>
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 <code>#USE *_IO</code> directive.
Availability:	All devices.
Requires:	Pin constants are defined in the devices .h file
Examples:	<pre>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 int16 i=PIN_B0;</pre>

```
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_x(), 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

Examples: `output_low(PIN_A0);`

```
Int16i=PIN_A1;
output_low(PIN_A1);
```

Example Files: `ex_sqw.c`

Also See: `input()`, `output_high()`, `output_float()`, `output_bit()`, `output_x()`, `#USE FIXED_IO`, `#USE FAST_IO`, `#USE STANDARD_IO`, General Purpose I/O

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);`

Example Files: None

Also See: `Input()`, `output_high()`, `output_low()`, `output_bit()`, `output_x()`

perror()

Syntax: `perror(string);`

Parameters: ***string*** is a constant string or array of characters (null terminated).

Returns: Nothing

Function: This function prints out to STDERR the supplied string and a description of the last system error (usually a math error).

Availability: All devices.

Requires: #USE RS232, #INCLUDE <errno.h>

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:	<pre>pid_get_result(PID_START_ONLY, ADCResult); while(pid_busy()); pid_get_result(PID_READ_ONLY, &PIDResult);</pre>
Example Files:	None
Also See:	setup_pid(), pid_write(), pid_get_result(), pid_read()

pid_get_result()

Syntax:	<pre>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);</pre>
Parameters:	<p>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:</p> <ul style="list-style-type: none"> · 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:

```
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 Files: None

Also See: 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: `pid_write(register, &input);`

Parameters: **register**- constant specifying which PID registers to write. 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_Z3
- PID_ADDR_K1
- PID_ADDR_K2

- PID_ADDR_K3

input -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.

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()`

pmp_address(address)

Syntax: `pmp_address (address);`

Parameters: ***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.

Returns: undefined

Function: 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.

Availability: Only the devices with a built in Parallel Port module.

Requires: Nothing.

Examples: `pmp_address(0x2100); // Sets up Address register to 0x2100`

Example None

Files:

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:	<code>result = pmp_output_full()</code>	<code>//PMP only</code>
	<code>result = pmp_input_full()</code>	<code>//PMP only</code>
	<code>result = pmp_overflow()</code>	<code>//PMP only</code>
	<code>result = pmp_error()</code>	<code>//EPMP only</code>
	<code>result = pmp_timeout()</code>	<code>//EPMP only</code>

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 Files:	None
Also See:	setup_pmp(), pmp_write(), pmp_read()

pmp_read()

Syntax:	<pre> 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 </pre>
----------------	--

Parameters:	<p>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)</p> <p>pointer- EPMP only, pointer to array to read data to.</p> <p>count- EPMP only, number of bytes to read. For pmp_read16() number of bytes must be even.</p>
--------------------	---

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:	<pre> 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. </pre>
------------------	--

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:

<code>pmp_write (data);</code>	<code>//Parallel Master Port</code>
<code>pmp_write8(address,data);</code>	<code>//Enhanced Parallel Master Port</code>
<code>pmp_write8(address,pointer,data);</code>	<code>//Enhanced Parallel Master Port</code>
<code>pmp_write16(address,data);</code>	<code>//Enhanced Parallel Master Port</code>
<code>pmp_write16(address,pointer,data);</code>	<code>//Enhanced Parallel Master Port</code>

Parameters:

data- The byte of data to be written.

address- EPMP only, address in EDS memory that is mapped to address from parallel port device to write data to or start writing data to. (All addresses in EDS memory are word aligned)

pointer- EPMP only, pointer to data to be written

count- EPMP only, number of bytes to write. For pmp_write16() number of bytes must be even.

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:	<pre> 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 </pre>
------------------	---

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:

```

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:	<code>port_a_pullups (FALSE) ;</code>
Example Files:	<code>ex_lcdkb.c, kbd.c</code>
Also See:	<code>input(), input_x(), output_float()</code>

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

Requires: `#INCLUDE <math.h>`

Examples: `area = pow (size,3.0) ;`

Example Files: None

Also See: None

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.

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). Note that format specifies do not work in ram band strings.

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:

c	Character
s	String or character
u	Unsigned int
d	Signed int
Lu	Long unsigned int
Ld	Long signed int
x	Hex int (lower case)
X	Hex int (upper case)
Lx	Hex long int (lower case)
LX	Hex long int (upper case)
f	Float with truncated decimal
g	Float with rounded decimal
e	Float in exponential format
w	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.

Example formats:

Specifier	Value=0x12	Value=0xfe
%03u	018	254
%u	18	254
%2u	18	*
%5	18	254
%d	18	-2
%x	12	fe
%X	12	FE
%4X	0012	00FE
%3.1w	1.8	25.4

* Result is undefined - Assume garbage.

Availability: All Devices

Requires: #USE RS232 (unless fname is used)

Examples:

```
byte x,y,z;
printf("HiThere");
printf("RTCCValue=>%2x\n\r",get_rtcc());
printf("%2u %X %4X\n\r",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

psp_output_full()

psp_input_full()

psp_overflow()

Syntax:

```
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:

```
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: **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:

```
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(), psp_output_full(), psp_input_full(), psp_overflow(), pmp_output_full(), pmp_input_full(), pmp_overflow().
See header file for device selected.

psp_write()

Syntax: `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()`, `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.

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 STDOUT (the last USE RS232).

Availability: All devices

Requires: #USE RS232

Examples:

```
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

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 of 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: #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

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:	<pre>puts(" ----- "); puts(" HI "); puts(" ----- ");</pre>
------------------	--

Example Files:	None
-----------------------	------

Also See:	printf(), gets(), RS232 I/O Overview
------------------	--------------------------------------

pwm_off()

Syntax:	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:	<pre>#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25) while(TRUE){ if(kbhit()){ c = getc(); if(c=='F') pwm_off(); } }</pre>
------------------	---

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:	<code>#USE PWM</code>
Examples:	<pre>#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25) while(TRUE){ if(kbhit()){ c = getc(); if(c=='O') pwm_on(); } }</pre>
Example Files:	None
Also See:	<code>#use_pwm</code> , <code>pwm_off()</code> , <code>pwm_set_duty_percent()</code> , <code>pwm_set_duty</code> , <code>pwm_set_frequency()</code>

pwm_set_duty()

Syntax:	<code>pwm_set_duty([stream],duty);</code>
Parameters:	<p>stream – optional parameter specifying the stream defined in <code>#USE PWM</code>.</p> <p>duty – an int16 constant or variable specifying the new PWM high time.</p>
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 <code>pwm_set_duty_percent()</code> , but requires you to know what the period of the PWM signal is.
Availability:	All devices.
Requires:	<code>#USE PWM</code>
Examples:	<pre>#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)</pre>
Example Files:	None
Also See:	<code>#use_pwm</code> , <code>pwm_on</code> , <code>pwm_off()</code> , <code>pwm_set_frequency()</code> , <code>pwm_set_duty_percent()</code>

pwm_set_duty_percent

Syntax:	<code>pwm_set_duty_percent([stream]), percent</code>
Parameters:	<p>stream – optional parameter specifying the stream defined in <code>#USE PWM</code>.</p> <p>percent- an int16 constant or variable ranging from 0 to 1000 specifying the</p>

Returns:	new PWM duty cycle, D is 0% and 1000 is 100.0%. Nothing.
Function:	To change the duty cycle of the PWM signal. Duty cycle percentage is based off the current frequency/period of the PWM signal.
Availability:	All devices.
Requires:	#USE PWM
Examples:	<pre>#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25) pwm_set_duty_percent(500); //set PWM duty cycle to 50%</pre>
Example Files:	None
Also See:	#use_pwm, pwm_on(), pwm_off(), pwm_set_frequency(), pwm_set_duty()

pwm_set_frequency

Syntax: `pwm_set_frequency([stream],frequency);`

Parameters:	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:	<pre>#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25) pwm_set_frequency(1000); //set PWM frequency to 1kHz</pre>
Example Files:	None
Also See:	#use_pwm, pwm_on(), pwm_off(), pwm_set_duty_percent, pwm_set_duty()

qei_get_count()

Syntax:	<code>value = qei_get_count([unit]);</code>
Parameters:	value - The 16-bit value of the position counter. unit - Optional unit number, defaults to 1.
Returns:	void
Function:	Reads the current 16-bit value of the position counter.
Availability:	Devices that have the QEI module.
Requires:	Nothing.
Examples:	<code>value = qei_get_counter();</code>
Example Files:	None
Also See:	<code>setup_qei()</code> , <code>qei_set_count()</code> , <code>qei_status()</code> .

qei_set_count()

Syntax:	<code>qei_set_count([unit,] value);</code>
Parameters:	value - The 16-bit value of the position counter. unit - Optional unit number, defaults to 1.
Returns:	void
Function:	Write a 16-bit value to the position counter.
Availability:	Devices that have the QEI module.
Requires:	Nothing.
Examples:	<code>qei_set_counter(value);</code>
Example Files:	None
Also See:	<code>setup_qei()</code> , <code>qei_get_count()</code> , <code>qei_status()</code> .

qei_status()

Syntax:	<code>status = qei_status([unit]);</code>
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:	<code>status = qei_status();</code>
Example Files:	None
Also See:	<code>setup_qei()</code> , <code>qei_set_count()</code> , <code>qei_get_count()</code> .

qsort()

Syntax:	<code>qsort (base, num, width, compare)</code>
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:	<code>#INCLUDE <stdlib.h></code>
Examples:	<code>int nums[5]={ 2,3,1,5,4};</code> <code>int compar(void *arg1,void *arg2);</code>

```
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();
```

Example Files: None

Also See: srand()

rcv_buffer_bytes()

Syntax: `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:

```
#USE_RS232(UART1,BAUD=9600,RECEIVE_BUFFER=10
0)
void main(void) {
    char c;
    if(rcv_buffer_bytes() > 10)
        c = getc();
}
```

Example Files: None

Also See: `_USE_RS232()`, `RCV_BUFFER_FULL()`, `TX_BUFFER_FULL()`, `TX_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:	<pre>#USE_RS232(UART1,BAUD=9600,RECEIVE_BUFFER=100) void main(void) { char c; if(rcv_buffer_full()) c = getc(); }</pre>
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])												
Parameters:	<i>mode</i> 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)												
Returns:	Either a 8 or 16 bit int depending on #DEVICE ADC= directive.												
Function:	<p>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:</p> <table border="1"> <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> </tbody> </table>	#DEVICE	10 bit	12 bit	ADC=8	00-FF	00-FF	ADC=10	0-3FF	0-3FF	ADC=11	x	x
#DEVICE	10 bit	12 bit											
ADC=8	00-FF	00-FF											
ADC=10	0-3FF	0-3FF											
ADC=11	x	x											

ADC=12	0-FFC	0-FFF
ADC=16	0-FFC0	0-FFF0

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;
setup_adc_ports(sAN0|sAN1, VSS_VDD);
setup_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\r", 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\r", value);
}
```

Example Files: ex_admm.c,

read_configuration_memory()

Syntax: 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: 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:

```
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 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:

```
#define LAST_VOLUME 10
volume = read_EEPROM (LAST_VOLUME);
```

Example Files: None

Also See: write_eeprom(), Data Eeprom Overview

read_extended_ram()

Syntax: 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:	<pre>unsigned int8 data[8]; read_extended_ram(1,0x0000,data,8);</pre>
Example Files:	None
Also See:	read_extended_ram(), Extended RAM Overview

read_program_memory()

Syntax:	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 . BDue 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:	<pre>char buffer[64]; read_external_memory(0x40000, buffer, 64);</pre>
Example Files:	None
Also See:	write_program_memory(), Program Eeprom Overview

read_high_speed_adc()

Syntax:	<code>read_high_speed_adc(pair,mode,result);</code>	<code>// Individual start and read or</code>
		<code>// read only</code>
	<code>read_high_speed_adc(pair,result);</code>	<code>// Individual start and read</code>
	<code>read_high_speed_adc(pair);</code>	<code>// Individual start only</code>
	<code>read_high_speed_adc(mode,result);</code>	<code>// Global start and read or</code>
		<code>// read only</code>
	<code>read_high_speed_adc(result);</code>	<code>// Global start and read</code>
	<code>read_high_speed_adc();</code>	<code>// Global start only</code>

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 INDIVIDUAL_SOFTWARE_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 GLOBAL_SOFTWARE_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 doing a read).

When using this function to perform a 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:

```
//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],
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 None

Files:

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**Also See:** setup_sd_adc(), set_sd_adc_calibration(), set_sd_adc_channel()

realloc()

Syntax: 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: #INCLUDE <stdlibm.h>

Examples:

```
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:	<code>release_io();</code>
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:	<pre>unsigned int16 restart; restart = restart_cause(); if(restart == RTC_FROM_DS) release_io();</pre>
Example Files:	None
Also See:	<code>sleep()</code>

reset_cpu()

Syntax:	<code>reset_cpu()</code>
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:	<pre>if(checksum!=0) reset_cpu();</pre>
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:

```
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:

	PCB/PCM	PCH
Enable/Disable	#fuses	setup_wdt()
Timeout time	setup_wdt()	#fuses
restart	restart_wdt()	restart_wdt()

Availability: All devices

Requires: #FUSES

Examples:

```
#fuses 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: 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;
rotate_left( &x, 1);
// x is now 0x0d
```

Example Files: None

Also See: rotate_right(), shift_left(), shift_right()

rotate_right()

Syntax: 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:

```
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: `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 structure date time.
Availability:	Devices that have the RTCC module.
Requires:	Nothing.
Examples:	<pre>rtc_alarm_write(&datetime);</pre>
Example Files:	None
Also See:	<code>rtc_read()</code> , <code>rtc_alarm_read()</code> , <code>rtc_alarm_write()</code> , <code>setup_rtc_alarm()</code> , <code>rtc_write()</code> , <code>setup_rtc()</code>

rtc_read()

Syntax:	<code>rtc_read(&datetime);</code>
Parameters:	<i>datetime</i> - 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 <code>rtc_time_t</code> .
Returns:	void
Function:	Reads the current value of Time and Date from the RTCC module and stores the structure date time.
Availability:	Devices that have the RTCC module.
Requires:	Nothing.
Examples:	<pre>rtc_read(&datetime);</pre>
Example Files:	<code>ex_rtcc.c</code>
Also See:	<code>rtc_read()</code> , <code>rtc_alarm_read()</code> , <code>rtc_alarm_write()</code> , <code>setup_rtc_alarm()</code> , <code>rtc_write()</code> , <code>setup_rtc()</code>

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 `date time`.

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()`

rtos_await()

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax: `rtos_await (expre)`

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:	<code>rtos_await(kbhit());</code>
------------------	-----------------------------------

Also See:	None
------------------	------

rtos_disable()

The RTOS is only included in the PCW, PCWH, and PCWHD software packages.

Syntax:	<code>rtos_disable (<i>task</i>)</code>
----------------	---

Parameters:	<i>task</i> 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 <code>rtos_enable()</code> . All tasks are enabled by default.
------------------	--

Availability:	All devices
----------------------	-------------

Requires:	<code>#USE RTOS</code>
------------------	------------------------

Examples:	<code>rtos_disable(toggle_green)</code>
------------------	---

Also See:	<code>rtos enable()</code>
------------------	----------------------------

rtos_enable()

The RTOS is only included in the PCW, PCWH, and PCWHD software packages.

Syntax:	<code>rtos_enable (<i>task</i>)</code>
----------------	--

Parameters:	<i>task</i> 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()`

rtos_msg_poll()

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax: `i = rtos_msg_poll()`

Parameters: None

Returns: An integer that specifies how many messages are in the queue.

Function: This function can only be used inside an RTOS task. This function returns the number of messages that are in the queue for the task that the `rtos_msg_poll()` function is used in.

Availability: All devices

Requires: #USE RTOS

Examples: `if(rtos_msg_poll())`

Also See: `rtos msg send()`, `rtos msg read()`

rtos_msg_read()

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax: `b = 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 `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_poll()`, `rtos_msg_send()`

rtos_msg_send()

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax: `rtos_msg_send(task, 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 *task*.

Availability: All devices

Requires: #USE RTOS

Examples:

```
if (kbhit())
{
    rtos_msg_send(echo, getc());
}
```

Also See: `rtos_msg_poll()`, `rtos_msg_read()`

rtos_overrun()

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax: `rtos_overrun([task])`

Parameters:	<i>task</i> 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 allotted execution time.
Availability:	All devices
Requires:	#USE RTOS(statistics)
Examples:	<code>rtos_overrun()</code>
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 <code>rtos_terminate()</code> is called.
Availability:	All devices
Requires:	#USE RTOS
Examples:	<code>rtos_run()</code>
Also See:	<code>rtos_terminate()</code>

rtos_signal()

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax:	rtos_signal (<i>sem</i>)
Parameters:	<i>sem</i> 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 <i>sem</i> to let waiting tasks know that a shared resource is available for use.
Availability:	All devices
Requires:	#USE RTOS
Examples:	<code>rtos_signal(uart_use)</code>
Also See:	<code>rtos wait()</code>

rtos_stats()

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax:	rtos_stats(<i>task</i>,&<i>stat</i>)
Parameters:	<p><i>task</i> is the identifier of a function that is being used as an RTOS task.</p> <p><i>stat</i> is a structure containing the following:</p> <pre> struct rtos_stas_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 = </pre>

	<pre>(ticks*hns_per_tick)/10 };</pre>
Returns:	Undefined
Function:	This function returns the statistic data for a specified <i>task</i> .
Availability:	All devices
Requires:	#USE RTOS(statistics)
Examples:	<pre>rtos_stats(echo, &stats)</pre>
Also See:	None

rtos_terminate()

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax:	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 <code>rtos_run()</code> call in the program. (This function causes <code>rtos_run()</code> to return.)
Availability:	All devices
Requires:	#USE RTOS
Examples:	<pre>rtos_terminate()</pre>
Also See:	rtos run()

rtos_wait()

The RTOS is only included in the PCW, PCWH and PCWHD software packages.

Syntax:	rtos_wait (sem)
----------------	------------------------

Parameters:	<i>sem</i> 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 <i>sem</i> to be greater than 0 (shared resource is available), then decrements <i>sem</i> 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
Requires:	#USE RTOS
Examples:	<code>rtos_wait(uart_use)</code>
Also See:	<code>rtos signal()</code>

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 <code>rtos_run()</code> . When the next task executes, it will start it's execution on the line of code after the <code>rtos_yield()</code> .
Availability:	All devices
Requires:	#USE RTOS
Examples:	<pre>void yield(void) { printf("Yielding...\r\n"); rtos_yield(); printf("Executing code after yield\r\n"); }</pre>

Also See: None

set_adc_channel()

set_adc_channel2()

Syntax: `set_adc_channel (chan [,neg])`
`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

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_analog_pins()

Syntax: `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:

- | | |
|-----------------|--|
| c | 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. |
| s | 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. |
| u | Matches an unsigned decimal integer. The corresponding argument shall be a pointer to an unsigned integer. |
| Lu | Matches a long unsigned decimal integer. The corresponding argument shall be a pointer to a long unsigned integer. |
| d | Matches a signed decimal integer. The corresponding argument shall be a pointer to a signed integer. |
| Ld | Matches a long signed decimal integer. The corresponding argument shall be a pointer to a long signed integer. |
| o | Matches a signed or unsigned octal integer. The corresponding argument shall be a pointer to a signed or unsigned integer. |
| Lo | Matches a long signed or unsigned octal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer. |
| x or X | Matches a hexadecimal integer. The corresponding argument shall be a pointer to a signed or unsigned integer. |
| Lx or LX | Matches a long hexadecimal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer. |
| i | Matches a signed or unsigned integer. The corresponding argument shall be a pointer to a signed or unsigned integer. |

Li	Matches a long signed or unsigned integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.
f,g or e	Matches a floating point number in decimal or exponential format. The corresponding argument shall be a pointer to a float.
[<p>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 -.</p> <p>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.</p>
n	<p>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.</p> <p>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.</p> <p>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.</p> <p>White-space characters are skipped, except for the conversion specifiers [, c or n, unless a white-space character precedes the [or c specifiers.</p>

Availability: All Devices

Requires: #USE RS232

Examples:

```
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 None

Files:

Also See: RS232 I/O Overview, getch(), putc(), printf()

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 CCPxRBL 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                // 200 and ending at
time 300
```

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 module, the times are either a 4-bit or 6-bit value, depending on the device's datasheet for the correct width.
Availability:	All devices with a COG module.
Examples:	<pre>set_cog_blanking(10,10);</pre>
Example Files:	None
Also See:	<code>setup_cog()</code> , <code>set_cog_phase()</code> , <code>set_cog_dead_band()</code> , <code>cog_status()</code> , <code>cog_re</code>

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 module, the times are either a 4-bit or 6-bit value, depending on the device's datasheet for the correct width.
Availability:	All devices with a COG module.
Examples:	<pre>set_cog_dead_band(16,32);</pre>
Example Files:	None
Also See:	<code>setup_cog()</code> , <code>set_cog_phase()</code> , <code>set_cog_blanking()</code> , <code>cog_status()</code> , <code>cog_re</code>

set_cog_phase()

Syntax: `set_cog_phase(rising_time);`
`set_cog_phase(falling_time, rising_time);`

Parameters: **falling time** - sets the falling edge phase time.

rising time - sets the rising edge phase time.

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 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`

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:	<pre>// 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);</pre>
Example Files:	None
Also See:	get_capture(), setup_compare(), Output Compare, PWM Overview

set_hspwm_duty()

Syntax:	setup_hspwm_duty(duty); set_hspwm_duty(unit, primary, [secondary]);
Parameters:	<p>duty - A 16-bit constant or variable to set the master duty cycle</p> <p>unit - The High Speed PWM unit to set.</p> <p>primary - A 16-bit constant or variable to set the primary duty cycle.</p> <p>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.</p>
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:	<pre>set_hspwm_duty(0x7FFF); //sets the High Speed PWM master duty cycle set_hspwm_duty(1, 0x3FFF); //sets unit 1's primary duty cycle</pre>
Example Files:	None
Also See:	setup_hspwm_unit(), set_hspwm_phase(), 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_hspwm_event()

set_hspwm_event_secondary()

Syntax: `set_hspwm_event(settings, compare_time);`
`set_shwpm_event_secondary(settings, compare_time); //if`
`available`

Parameters: **settings** - special event timer setting or'd with a value from 1 to 16 to set the prescaler. The following are the settings available for the special event time:

- HSPWM_SPECIAL_EVENT_INT_ENABLED
- HSPWM_SPECIAL_EVENT_INT_DISABLED

compare_time - the compare time for the special event to occur.

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: `set_hspwm_event(HSPWM_SPECIAL_EVENT_INT_ENABLED, 0x1000);`

Example Files: None

Also See: `setup_hspwm_unit()`, `set_hspwm_phase()`, `set_hspwm_duty()`,
`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_hspwm_override()

Syntax: `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 uoverride settings.

Availability: Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

Requires: None

Examples: `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: `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: `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_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.

Availability: Devices that have the motor control PWM unit.

Requires: None

Examples: `set_motor_pwm_duty(1,0,0x55); // Sets the PWM1 Unit a duty cycle value`

Example Files:	None
Also See:	<code>get_motor_pwm_count()</code> , <code>set_motor_pwm_event()</code> , <code>set_motor_unit()</code> , <code>setup_motor_pwm()</code>

set_motor_pwm_event()

Syntax:	<code>set_motor_pwm_event(pwm,time);</code>
Parameters:	<i>pwm</i> - Defines the pwm module used. <i>time</i> - 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:	<code>set_motor_pwm_event(pwm,time);</code>
Example Files:	None
Also See:	<code>get_motor_pwm_count()</code> , <code>setup_motor_pwm()</code> , <code>set_motor_unit()</code> , <code>set_motor_pwm_duty()</code> ;

set_motor_unit()

Syntax:	<code>set_motor_unit(pwm,unit,options, active_deadtime, inactive_deadtime);</code>
Parameters:	<i>pwm</i> - Defines the pwm module used <i>Unit</i> - This will select Unit A or Unit B <i>options</i> - 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:

```
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: `set_nco_inc_value(value);`

Parameters: **value**- 16-bit value to set the NCO increment registers to (0 - 65535)

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_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)

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 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 Enables/Disables open-drain output capability on port pins. Not all ports or port pins have open-drain capability, refer to devices datasheet for port and pin availability.

Availability On device that have open-drain capability.

Examples: **set_open_drain_b(0x0001); //enables open-drain output on PIN_B0, disable on all //other port B pins.**

Example Files: 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.

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:

```
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 None

Files:

Also See: None

set_pwm1_duty()
set_pwm2_duty()
set_pwm3_duty()
set_pwm4_duty()
set_pwm5_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:

- $\text{duty cycle} = \text{value} / [4 * (\text{PRx} + 1)]$

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:

- $\text{duty cycle} = \text{value} / (\text{CCPxPRL} + 1)$

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:

```
// 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(), get_captures32_ccpX()

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:	<pre>// 20 mhz clock, no prescaler, set timer 0 // to overflow in 35us set_timer0(81); // 256-(.000035/(4/20000000))</pre>

Example Files: ex_patg.c

Also See: set_timer1(), get_timerX() Timer0 Overview, Timer1 Overview, 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;
```

Example None

Files:

Also See: setup_sd_adc(), read_sd_adc(), set_sd_adc_channel()

set_sd_adc_channel()

Syntax: 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 None

Files:

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:

```
// 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 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:	<pre>if(get_timer45() == THRESHOLD) set_timer(THRESHOLD + 0x1000); //skip those timer values</pre>
Example Files:	None
Also See:	Timer Overview, setup_timerX(), get_timerXY(), set_timerX(), set_timerXY()

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_timer_ccp1()
set_timer_ccp2()
set_timer_ccp3()
set_timer_ccp4()
set_timer_ccp5()

Syntax: **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.

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_ccpX_compare_time(), get_capture_ccpX(), set_timer_period_ccpX(), get_timer_ccpx(), get_captures32_ccpX()

set_timer_period_ccp1()
set_timer_period_ccp2()
set_timer_period_ccp3()
set_timer_period_ccp4()
set_timer_period_ccp5()

Syntax: set_timer_period_ccpx(time);
set_timer_period_ccpx(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                       //upper timer period to
2000
```

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

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:

```
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: `set_uart_speed (baud, [stream, clock])`

Parameters: *baud* is a constant representing the number of bits per second.

stream is an optional stream identifier.
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 run-time.

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;
}
```

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()`

setup_adc(mode)

setup_adc2(mode)

Syntax: `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:

```
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()

setup_adc_ports2()

Syntax: **setup_adc_ports (value)**
 setup_adc_ports (ports, [reference])
 setup_adc_ports (ports, [reference])

Parameters: **value** - a constant defined in the devices .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: Only available on devices with built in analog to digital converters

Requires: Constants are defined in the devices .h file.

Examples:

```
// 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 used for voltage references
setup_adc_ports(sAN0|sAN1, VREF_VDD);
```

Example ex_admm.c

Files:**Also See:** `setup_adc()`, `read_adc()`, `set_adc_channel()`, ADC 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:****Also See:** `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: `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:

```
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: **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_pwm_duty() is provided for setting the duty cycle when in PWM 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, CCP_COMPARE_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:

```
setup_ccp1(CCP_CAPTURE_FE);
setup_ccp1(CCP_COMPARE_TOGGLE);
setup_ccp1(CCP_PWM);
```

Example Files: `ex_pwm.c`, `ex_ccmp.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()`, `get_captures32_ccpX()`

setup_clc1() setup_clc2() setup_clc3() setup_clc4()

Syntax:

```
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: `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`

Example Files:

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);
```

Example Files: None

Also See: `set_compare_time()`, `set_pwm_duty()`, `setup_capture()`, Output Compare / PWM Overview

setup_crc(mode)

Syntax: `setup_crc(polynomial 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 Modul Only), to enable little-endian shift direction:

· `CRC_LITTLE_ENDIAN`

Returns:	undefined
Function:	Configures the CRC engine register with the polynomial
Availability:	Only the devices with built in CRC module
Requires:	Nothing
Examples:	<pre>setup_crc (12, 5); // CRC Polynomial is X¹² + X⁵ + 1 setup_crc(16, 15, 3, 1); // CRC Polynomial is X¹⁶ + X¹⁵ + X³ + X¹ + 1</pre>
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

steering- 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 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);
```

Example Files: None

Also See: `set_cog_dead_band()`, `set_cog_phase()`, `set_cog_blanking()`, `cog_stat`

setup_crc()

Syntax: `setup_crc(polynomial 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:	<pre>setup_crc(12, 5); // CRC Polynomial is x¹²+x⁵+1 setup_crc(16, 15, 3, 1); // CRC Polynomial is x¹⁶+x¹⁵+x³+x¹+1</pre>
Example Files:	None
Also See:	<code>crc_init()</code> , <code>crc_calc()</code> , <code>crc_calc8()</code>

setup_cwg()

Syntax: `setup_cwg(mode,shutdown,dead_time_rising,dead_time_falling)`

Parameters:	<p>mode- the setup of the CWG module. See the device's .h file for all options. Some typical options include:</p> <ul style="list-style-type: none"> • CWG_ENABLED • CWG_DISABLED • CWG_OUTPUT_B • CWG_OUTPUT_A <p>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:</p> <pre>CWG_AUTO_RESTART CWG_SHUTDOWN_ON)COMP1 CWG_SHUTDOWN_ON_FLT CWG_SHUTDOWN_ON_CLC2</pre> <p>dead_time_rising- value specifying the dead time between A and B on the rising edge. (0-63)</p> <p>dead_time_falling- value specifying the dead time between A and B on the falling edge. (0-63)</p>
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:	<pre>setup_cwg(CWG_ENABLED CWG_OUTPUT_A CWG_OUTPUT_B CWG_INPUT_PWM1,CWG_SHUTDOWN_ON_FLT,60,30);</pre>
Example Files:	None
Also See:	<code>cwg_status()</code> , <code>cwg_restart()</code>

setup_dac()

Syntax:	setup_dac(mode); setup_dac(mode, divisor);
Parameters:	<p>mode- The valid options vary depending on the device. See the devices .h file for all options. Some typical options include:</p> <ul style="list-style-type: none"> · DAC_OUTPUT <p>divisor- Divides the provided clock</p>
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:	<pre>setup_dac(DAC_VDD DAC_OUTPUT); dac_write(value); setup_dac(DAC_RIGHT_ON, 5);</pre>
Example Files:	None
Also See:	<code>dac_write()</code> , DAC Overview, See header file for device selected

setup_dci()

Syntax:	setup_dci(configuration, data size, rx config, tx config, sample rate);
Parameters:	<p><i>configuration</i> - 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:</p> <ul style="list-style-type: none"> · CODEC_MULTICHANNEL · CODEC_I2S· CODEC_AC16 · CODEC_AC20· JUSTIFY_DATA· DCI_MASTER · DCI_SLAVE· TRISTATE_BUS· MULTI_DEVICE_BUS · SAMPLE_FALLING_EDGE· SAMPLE_RISING_EDGE · DCI_CLOCK_INPUT· DCI_CLOCK_OUTPUT <p><i>data size</i> – Specifies the size of frames and words in the transmission:</p> <ul style="list-style-type: none"> · DCI_xBIT_WORD: x may be 4 through 16 · DCI_xWORD_FRAME: x may be 1 through 16 · DCI_xWORD_INTERRUPT: x may be 1 through 4 <p><i>rx config</i>- Specifies which words of a given frame the DCI module will receive (commonly used for a multi-channel, shared bus situation)</p> <ul style="list-style-type: none"> · RECEIVE_SLOTx: x May be 0 through 15 · RECEIVE_ALL· RECEIVE_NONE <p><i>tx config</i>- Specifies which words of a given frame the DCI module will transmit on.</p> <ul style="list-style-type: none"> · TRANSMIT_SLOTx: x May be 0 through 15 · TRANSMIT_ALL · TRANSMIT_NONE <p><i>sample rate</i>-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.</p>
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:	<code>dci_initialize((I2S_MODE DCI_MASTER DCI_CLOCK_OUTPUT </code>

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
- ADC_HALT_IDLE – The ADC will not run when PIC is idle.

Returns: Undefined

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.

Availability: Only on dsPIC33FJxxGSxxx devices.

Requires: Constants are define in the device .h file.

Examples:

```
setup_high_speed_adc_pair(0, INDIVIDUAL_SOFTWARE_TRIGGER);
setup_high_speed_adc(ADC_CLOCK_DIV_4);
read_high_speed_adc(0, START_AND_READ, result);
setup_high_speed_adc(ADC_OFF);
```

Example Files: None

Also See: setup_high_speed_adc_pair(), read_high_speed_adc(), high_speed_adc_done()

setup_high_speed_adc_pair()

Syntax: 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:

```
setup_high_speed_adc_pair(0, INDIVIDUAL_SOFTWARE_TRIGGER);
setup_high_speed_adc_pair(1, GLOBAL_SOFTWARE_TRIGGER);
setup_high_speed_adc_pair(2, 0) - sets AN4 and AN5 as digital pins.
```

Example Files: None

Also See: setup_high_speed_adc(), read_high_speed_adc(), high_speed_adc_done()

setup_hspwm_blanking()

Syntax: **setup_hspwm_blanking(unit, settings, delay);**

Parameters: **unit** - The High Speed PWM unit to set.

start_delay - Optional value from 0 to 63 specifying then umber 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.

strigger_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.

Availability: Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

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_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(), set_hspwm_override(),
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.

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_PWML_TRIGGERS_LE_BLANKING
- HSPWM_FE_PWML_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: `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_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.

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.

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: `setup_hspwm_unit(1, HSPWM_ENABLE|HSPWM_COMPLEMENTARY, 100, 100);`

Example Files: None

Also See: `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(), 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, dsPIC33EPxxxMUxxx, 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, dsPIC33EPxxxMUxxx, 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 post-scale.

prescale- This will select the PWM timebase prescale setting

	<i>postscale</i> - This will select the PWM timebase postscale setting
Returns:	void
Function:	Configures the motor control PWM module
Availability:	Devices that have the motor control PWM unit.
Requires:	None
Examples:	<pre>setup_motor_pwm(1,MPWM_FREE_RUN MPWM_SYNC_OVERRIDES, timebase);</pre>
Example Files:	None
Also See:	get motor pwm count(), set motor pwm event(), set motor unit(), set motor pwm duty();

setup_oscillator()

Syntax: **setup_oscillator(*mode*, *target* [,*source*] [,*divide*])**

Parameters:	<p>Mode is one of:</p> <ul style="list-style-type: none"> • OSC_INTERNAL • OSC_CRYSTAL • OSC_CLOCK • OSC_RC • OSC_SECONDARY <p>Target is the target frequency to run the device it.</p> <p>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.</p> <p>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.</p>
--------------------	--

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:	<pre>setup_oscillator(OSC_CRYSTAL, 4000000, 16000000); setup_oscillator(OSC_INTERNAL, 29480000);</pre>
Example Files:	None
Also See:	setup_wdt(), Internal Oscillator Overview

setup_pid()

Syntax:	setup_pid(,pde.[mode],[K1],[K2],[K3]);
Parameters:	<p>mode- the setup of the PID module. The options for setting up the module are defined in the device's header file as:</p> <ul style="list-style-type: none"> · 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 <p>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:</p> <ul style="list-style-type: none"> · $K1 = Kp + Ki * T + Kd/T$ <p>When in one of the ADD_MULTIPLY modes K1 is the multiple value.</p> <p>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</p>

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: `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(options,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` //read/write cycle //Master Mode 1
- `PAR_WAITE4` //4 Tcy Wait for data hold after

```
// strobe
```

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:

```
setup_psp(PAR_ENABLE |          //Sets up Master mode with
address
PAR_MASTER_MODE_1 | PAR_      //lines PMA0:PMA7
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_power_pwm_pins()

Syntax: setup_power_pwm_pins(module0,module1,module2,module3)

Parameters: For each module (two pins) specify:
PWM_PINS_DISABLED, PWM_ODD_ON, PWM_BOTH_ON,
PWM_COMPLEMENTARY

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:	<pre> setup_power_pwm_pins(PWM_PINS_DISABLED, PWM_PINS_DISABLED, PWM_PINS_DISABLED, PWM_PINS_DISABLED); setup_power_pwm_pins(PWM_COMPLEMENTARY, PWM_COMPLEMENTARY, PWM_PINS_DISABLED, PWM_PINS_DISABLED); </pre>
Example Files:	None
Also See:	<pre> setup_power_pwm(), set_power_pwm_override(),set_power_pwmX_duty() </pre>

setup_psp(option,address_mask)

Syntax:	<pre> setup_psp (options,address_mask); setup_psp(options); </pre>
Parameters:	<p>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:</p> <ul style="list-style-type: none"> · 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 //read/write cycle //4 Tcy Wait for data hold after · //strobe <p>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 PMA0:PMA31 (EPMP only).</p>
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|           //Sets up legacy slave
          PAR_STOP_IN_IDLE,0x00FF ); //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:

```
setup_pwm1 (PWM_ENABLED|PWM_OUTPUT) ;
```

Example	None
Files:	
Also See:	set_pwm_duty()

setup_qei()

Syntax: `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: `setup_qei(QEI_MODE_X2|QEI_TIMER_INTERNAL,QEI_FILTER_DIV_2, QEI_FORWARD);`

Example Files: None

Also See: `qei_set_count()` , `qei_get_count()` , `qei_status()`

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, settings 2, 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`
- 12 `SDADC_OSR_1024`
- 13 `SDADC_CLK_SYSTEM`

Returns: Nothing

Function: To setup the Sigma-Delta Analog to Digital Converter (SD ADC) module.

Availability: Only devices with a SD ADC module.

Examples: `setup_sd_adc(SDADC_ENABLED | SDADC_DITHER_LOW,`
`SDADC_CHOPPING_ENABLED |`
`SDADC_INT_EVERY_5TH_SAMPLE |`


```
SDADC_RES_UPDATED_EVERY_INT, SDADC_CLK_SYSTEM |
SDADC_CLOCK_DIV_4);
```

Example None

Files:

Also See: set_sd_adc_channel(), read_sd_adc(), set_sd_adc_calibration()

setup_smtx()

Syntax: setup_smtx1(mode,[period]);
setup_smtx2(mode,[period]);

Parameters: **mode** - The setup of the SMT module. See the device's .h file for all 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 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: setup_smtx1(SMT_ENABLED | SMT_MODE_PERIOD_DUTY_CYCLE_ACQ |
SMT_REPEAT_DATA_ACQ_MODE | SMT_CLK_FOSC);

Example None

Files:

Also See: smtx_status(), smtx_start(), smtx_stop(), smtx_update(), smtx_read(), smtx_write()

setup_spi() setup_spi2()

Syntax: setup_spi(mode)
setup_spi2(mode)

Parameters:	<p><i>mode</i> may be:</p> <ul style="list-style-type: none"> • 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:	<p>Configures the hardware SPI™ module.</p> <ul style="list-style-type: none"> • 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.
Examples:	<code>setup_spi(SPI_MASTER SPI_L_TO_H SPI_DIV_BY_16);</code>
Example Files:	ex_spi.c
Also See:	spi_write(), spi_read(), spi_data_is_in(), SPI Overview

setup_timerx()

Syntax:	<p>setup_timerX(<i>mode</i>) setup_timerX(<i>mode,period</i>)</p>
Parameters:	<p>Mode is a bit-field comprised of the following configuration constants:</p> <ul style="list-style-type: none"> • 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 SOSC1/SOSCO pins
- **T1_EXTERNAL_SYNC**: Uses a clock source that is connected to the SOSC1/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.
- **T1_EXTERNAL_RTC**: Uses a low power clock source connected to the SOSC1/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 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 an interrupt that will be fired when that
timer overflows*/
setup_timer4(TMR_32_BIT); //use get_timer45() to get the
timer value
enable_interrupts(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_A()

Syntax: `setup_timer_A (mode);`

Parameters: *mode* values may be:

- TA_OFF, TA_INTERNAL, TA_EXT_H_TO_L, TA_EXT_L_TO_H
- TA_DIV_1, TA_DIV_2, TA_DIV_4, TA_DIV_8, TA_DIV_16, TA_DIV_32, TA_DIV_64, TA_DIV_128, TA_DIV_256
- constants from different groups may be or'ed together with |.

Returns: undefined

Function: sets up Timer A.

Availability: This function is only available on devices with Timer A hardware.

Requires: Constants are defined in the device's .h file.

Examples:

```
setup_timer_A(TA_OFF);
setup_timer_A(TA_INTERNAL | TA_DIV_256);
setup_timer_A(TA_EXT_L_TO_H | TA_DIV_1);
```

Example Files: none

Also See: `get_timerA()`, `set_timerA()`, TimerA Overview

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:	<pre>setup_timer_B(TB_OFF); setup_timer_B(TB_INTERNAL TB_DIV_256); setup_timer_B(TA_EXT_L_TO_H TB_DIV_1);</pre>
Example Files:	none
Also See:	get_timerB(), set_timerB(), TimerB Overview

setup_timer_0()

Syntax:	setup_timer_0 (mode)
Parameters:	<p>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</p> <p>RTCC_DIV_2, RTCC_DIV_4, RTCC_DIV_8, RTCC_DIV_16, RTCC_DIV_32, RTCC_DIV_64, RTCC_DIV_128, RTCC_DIV_256</p> <p>PIC18XXX only: RTCC_OFF, RTCC_8_BIT</p> <p>One constant may be used from each group or'ed together with the operator.</p>
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:	<pre>setup_timer_0 (RTCC_DIV_2 RTCC_EXT_L_TO_H);</pre>
Example Files:	

Also See: `get_timer0()`, `set_timer0()`, `setup_counters()`

setup_timer_1()

Syntax: `setup_timer_1 (mode)`

Parameters: *mode* values may be:

- `T1_DISABLED`, `T1_INTERNAL`, `T1_EXTERNAL`, `T1_EXTERNAL_SYNC`
- `T1_CLK_OUT`
- `T1_DIV_BY_1`, `T1_DIV_BY_2`, `T1_DIV_BY_4`, `T1_DIV_BY_8`
- constants from different groups may be or'ed together with `|`.

Returns: undefined

Function: 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.

With an internal clock at 20mhz and with the `T1_DIV_BY_8` mode, the timer will increment every 1.6us. It will overflow every 104.8576ms.

Availability: This function is only available on devices with timer 1 hardware.

Requires: Constants are defined in the devices .h file.

Examples:

```
setup_timer_1 ( T1_DISABLED );
setup_timer_1 ( T1_INTERNAL | T1_DIV_BY_4 );
setup_timer_1 ( T1_INTERNAL | T1_DIV_BY_8 );
```

Example Files:

Also See: `get_timer1()`, Timer1 Overview

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.

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.`

Example Files:

Also See: `get_timer4(), set_timer4()`

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 ($V_{dd} * \text{value} / 24$)
- VREF_HIGH ($V_{dd} * \text{value} / 32 + V_{dd}/4$)
- 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`
- `WDT_64MS`
- `WDT_1S`
- `WDT_16S`

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:

```
setup_wdt(WDT_ON);
```

Example Files: `ex_wdt.c`

Also See: Internal Oscillator Overview

setup_zdc()

Syntax: `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: `setup_zcd(ZCD_ENABLE|ZCD_INT_H_TO_L);`

Example Files: None

Also See: `zcd_status()`

shift_left()

Syntax: `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:

```
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 ex_extee.c, 9356.c

Files:

Also See: shift_right(), rotate_right(), rotate_left(),

shift_right()

Syntax: 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.

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:

```

// 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 ex_extee.c, 9356.c

Files:**Also See:** shift_left(), rotate_right(), rotate_left(),

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_interrupts(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(<i>time</i>)
Parameters:	<i>time</i> 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:	<pre>while (TRUE) { if (input(PIN_A1)) //do something else sleep_ulpwu(10); //cap will be charged for 10us, //then goto sleep }</pre>
Example Files:	None
Also See:	#USE DELAY

smtx_read()

Syntax:	value_smt1_read(which); value_smt2_read(which);
Parameters:	<p>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:</p> <pre>SMT_CAPTURED_PERIOD_REG SMT_CAPTURED_PULSE_WIDTH_REG SMT_TMR_REG SMT_PERIOD_REG</pre>

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:	unsigned int32 Period; <code>Period = smt1_read(SMT_CAPTURED_PERIOD_REG);</code>
Example Files:	None
Also See:	<code>smtx_status()</code> , <code>stmx_start()</code> , <code>smtx_stop()</code> , <code>smtx_update()</code> , <code>smtx_reset_timer()</code> , <code>setup_SMTx()</code> , <code>smtx_write()</code>

smtx_reset_timer()

Syntax:	<code>smt1_reset_timer();</code> <code>smt2_reset_timer();</code>
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:	<code>smt1_reset_timer();</code>
Example Files:	None
Also See:	<code>setup_smtx()</code> , <code>stmx_start()</code> , <code>smtx_stop()</code> , <code>smtx_update()</code> , <code>smtx_status()</code> , <code>smtx_read()</code> , <code>smtx_write()</code>

smtx_start()

Syntax: `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_read()`, `smtx_write()`

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();`

Example Files: None

Also See: `setup_smtx()`, `stmx_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

Also See: `smtx_status()`, `smtx_start()`, `setup_smtx()`, `smtx_update()`,
`smtx_reset_timer()`,
`smtx_read()`, `smtx_write()`

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:	<code>smt1_write(SMT_PERIOD_REG, 0x100000000);</code>
Example Files:	None
Also See:	<code>smtx_status()</code> , <code>smtx_start()</code> , <code>setup_smtx()</code> , <code>smtx_update()</code> , <code>smtx_reset_timer()</code> , <code>smtx_read()</code> , <code>setup_smtx()</code>

smtx_update()

Syntax:	<code>smt1_update(which);</code> <code>smt2_update(which);</code>
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: <code>SMT_CAPTURED_PERIOD_REG</code> <code>SMT_CAPTURED_PULSE_WIDTH_REG</code>
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:	<code>smt1_update(SMT_CAPTURED_PERIOD_REG);</code>
Example Files:	None
Also See:	<code>setup_smtx()</code> , <code>smtx_start()</code> , <code>smtx_stop()</code> , <code>smtx_status()</code> , <code>smtx_reset_timer()</code> , <code>smtx_read()</code> , <code>smtx_write()</code>

spi_data_is_in() spi_data_is_in2()

Syntax:	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:	<pre>(!spi_data_is_in() && input(PIN_B2)); if(spi_data_is_in()) data = spi_read();</pre>
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:	<pre>#use spi(MATER, SPI1, baud=1000000, mode=0,</pre>

```

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 than 8 bits for PCM and PCH device, or more than 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:	<p>Return a value read by the SPI. If a value is passed to the <code>spi_read_16()</code> the data will be clocked out and the data received will be returned. If no data is ready, <code>spi_read_16()</code> will wait for the data is a SLAVE or return the last DATA clocked in from <code>spi_write_16()</code>.</p> <p>If this device is the MASTER then either do a <code>spi_write_16(data)</code> followed by a <code>spi_read_16()</code> or do a <code>spi_read_16(data)</code>. These both do the same thing and will generate a clock. If there is no data to send just do a <code>spi_read_16(0)</code> to get the clock.</p> <p>If this device is a slave then either call <code>spi_read_16()</code> to wait for the clock and data or use <code>spi_data_is_in()</code> to determine if data is ready.</p>
Availability:	This function is only available on devices with SPI hardware.
Requires:	NThat the option <code>SPI_MODE_16B</code> be used in <code>setup_spi()</code> function, or that the option <code>XFER16</code> be used in <code>#use SPI</code>
Examples:	<pre>data_in = spi_read_16(out_data);</pre>
Example Files:	None
Also See:	<code>spi_read()</code> , <code>spi_write()</code> , <code>spi_write_16()</code> , <code>spi_data_is_in()</code> , SPI Overview

spi_speed

Syntax:	<pre>spi_speed(baud); spi_speed(stream,baud); spi_speed(stream,baud,clock);</pre>
Parameters:	<p>stream – is the SPI stream to use as defined in the <code>STREAM=name</code> option in <code>#USE SPI</code>.</p> <p>band- the band rate to set the SPI module to</p> <p>clock- the current clock rate to calculate the band rate with. If not specified it uses the value specified in <code>#use delay ()</code>.</p>
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:	<code>#USE SPI</code>
Examples:	<pre>spi_speed(250000); spi_speed(SPI1_MODE0, 250000);</pre>

Example Files:	<code>spi_speed(SPI1_MODE0, 125000, 8000000);</code> None
Also See:	<code>#USE SPI, spi_xfer(), spi_xfer_in(), spi_prewrite(), spi_init()</code>

`spi_write()`

`spi_write2()`

`spi_write3()`

`spi_write4()`

Syntax:	<code>spi_write([wait],value);</code> <code>spi_write2([wait],value);</code> <code>spi_write3([wait],value);</code> <code>spi_write4([wait],value);</code>
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. <code>spi_read()</code> may be used to read the buffer.
Availability:	This function is only available on devices with SPI hardware.
Requires:	Nothing
Examples:	<code>spi_write(data_out);</code> <code>data_in = spi_read();</code>
Example Files:	<code>ex_spi.c</code>
Also See:	<code>spi_read(), spi_data_is_in(), SPI Overview, spi_write_16(), spi_read_16()</code>

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:

```
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
```

Example Files: None

Also See: #USE SPI

SPII_XFER_IN()

Syntax:

```
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 first.
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:	<pre>data_in = spi_xfer_in();</pre>
Example Files:	ex_spi_slave.c
Also See:	#USE SPI, spi_xfer(), spi_prewrite(), spi_init(), spi_speed()

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 specifiers 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 Files: None

Also See: printf()

sqrt()

Syntax:	result = sqrt (value)
Parameters:	value is any float type
Returns:	Returns a floating point value with a precision equal to value
Function:	<p>Computes the non-negative square root of the float value x. If the argument is negative, the behavior is undefined.</p> <p>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.</p> <p>Domain error occurs in the following cases: sqrt: when the argument is negative</p>
Availability:	All devices.
Requires:	#INCLUDE <math.h>
Examples:	<pre>distance = sqrt(pow((x1-x2),2)+pow((y1-y2),2));</pre>
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:	<pre>srand(10); I=rand();</pre>
Example Files:	None
Also See:	rand()

STANDARD STRING FUNCTIONS()

memchr() memcmp()

strcat() strchr() strcmp()

strcoll() strcspn() strerror()

stricmp() strlen() strlwr()

strncat() strncmp() strncpy()

strpbrk() strrchr() strspn() strstr() strxfrm()

Syntax:

ptr=strcat (s1, s2)	Concatenate s2 onto s1
ptr=strchr (s1, c)	Find c in s1 and return &s1[i]
ptr=strrchr (s1, c)	Same but search in reverse
result=strcmp (s1, s2)	Compare s1 to s2
ireult=strncmp (s1, s2, n)	Compare s1 to s2 (n bytes)
ireult=strcmp (s1, s2)	Compare and ignore case
ptr=strncpy (s1, s2, n)	Copy up to n characters s2->s1
ireult=strcspn (s1, s2)	Count of initial chars in s1 not in s2
ireult=strspn (s1, s2)	Count of initial chars in s1 also in s2
ireult=strlen (s1)	Number of characters in s1
ptr=strlwr (s1)	Convert string to lower case
ptr=strpbrk (s1, s2)	Search s1 for first char also in s2
ptr=strstr (s1, s2)	Search for s2 in s1
ptr=strncat(s1,s2, n)	Concatenates up to n bytes of s2 onto s1
ireult=strcoll(s1,s2)	Compares s1 to s2, both interpreted as appropriate to the current locale.
res=strxfrm(s1,s2,n)	Transforms maximum of n characters of s2 and places them in s1, such that

	strcmp(s1,s2) will give the same result as strcoll(s1,s2)
ireport=memcmp(m1,m2,n)	Compare m1 to m2 (n bytes)
ptr=memchr(m1,c,n)	Find c in first n characters of m1 and return &m1[j]
ptr=strerror(ernum)	Maps the error number in ernum to an error message string. The parameters 'ernum' is an unsigned 8 bit int. Returns a pointer to the string.

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
ireport 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:

```
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
```

Example Files: ex_str.c

Also See: strcpy(), strtok()

strcpy() strcpy()

Syntax: `strcpy (dest, src)`
`strcpy (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:

```
char string[10], string2[10];
.
.
.
strcpy (string, "Hi There");

strcpy(string2, string);
```

Example Files: ex_str.c

Also See: strxxxx()

strtod() strtod() strtod48()

Syntax: `result=strtod(nptr,& endptr)`
`result=strtod(nptr,& endptr)`
`result=strtod48(nptr,& endptr)`

Parameters: *nptr* and *endptr* are strings

Returns: strtod returns a double precision floating point number.
 strtod returns a single precision floating point number.
 strtod48 returns an extended precision floating point number.
 returns the converted value in result, if any. If no conversion could be

performed, zero is returned.

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 `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:

```
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:

```
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: **result=strtol(*nptr*,& *endptr*, *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:	<pre>signed long result; char str[9]="123hello"; char *ptr; result=strtol(str,&ptr,10); //result is 123 and ptr is "hello"</pre>
Example Files:	None
Also See:	strtod(), strtoul()

strtoul()

Syntax:	result=strtoul(<i>nptr</i>,<i>endptr</i>, <i>base</i>)
Parameters:	<i>nptr</i> and <i>endptr</i> are strings pointers and <i>base</i> 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 <i>nptr</i> to a long int representation in some radix determined by the value of <i>base</i> . The part of the string after conversion is stored in the object pointed to <i>endptr</i> , provided that <i>endptr</i> is not a null pointer. If <i>nptr</i> is empty or does not have the expected form, no conversion is performed and the value of <i>nptr</i> is stored in the object pointed to by <i>endptr</i> , provided <i>endptr</i> is not a null pointer.
Availability:	All devices.
Requires:	STDLIB.H must be included
Examples:	<pre>long result; char str[9]="123hello"; char *ptr; result=strtoul(str,&ptr,10); //result is 123 and ptr is "hello"</pre>

Example Files:	None
Also See:	strtol(), strtod()

swap()

Syntax:	swap (<i>Ivalue</i>) result = swap(<i>Ivalue</i>)
Parameters:	<i>Ivalue</i> 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);
Availability:	All devices.
Requires:	Nothing
Examples:	<pre>x=0x45; swap(x); //x now is 0x54 int x = 0x42; int result; result = swap(x); // result is 0x24;</pre>
Example Files:	None
Also See:	rotate_right(), rotate_left()

tolower() toupper()

Syntax:	result = tolower (<i>cvalue</i>) result = toupper (<i>cvalue</i>)
Parameters:	<i>cvalue</i> is a character

Returns:	An 8 bit character
Function:	<p>These functions change the case of letters in the alphabet.</p> <p>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.</p>
Availability:	All devices.
Requires:	Nothing
Examples:	<pre>switch(toupper(getc())) { case 'R' : read_cmd(); break; case 'W' : write_cmd(); break; case 'Q' : done=TRUE; break; }</pre>
Example Files:	ex_str.c
Also See:	None

touchpad_getc()

Syntax:	input = TOUCHPAD_GETC();
Parameters:	None
Returns:	char (returns corresponding ASCII number if “input” declared as int)
Function:	<p>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”.</p> <p>Note: Until a CSM or CTMU pin is read by firmware as active, this instruction will cause the microcontroller to stall.</p>
Availability:	All PIC's with a CSM or CTMU Module
Requires:	#USE TOUCHPAD (options)
Examples:	<pre>//When the pad connected to PIN_B0 is activated, store the letter 'A' #USE TOUCHPAD (PIN_B0='A')</pre>

```
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

Also See: #USE TOUCHPAD, touchpad_state()

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: // When the pad connected to PIN_B0 is activated, store the letter 'A'

```
#USE TOUCHPAD (PIN_B0='A')
void main(void){
    char c;
    enable_interrupts(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()

touchpad_state()

Syntax: TOUCHPAD_STATE (*state*);

Parameters: *state* is a literal 0, 1, or 2.

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'
}
//if PIN_D5 is pressed, c will get value '5'
```

Example Files: None

Also See: #USE TOUCHPAD, touchpad_getc(), touchpad_hit()

tx_buffer_available()

Syntax: `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 has a circular buffer, so be sure to check to make sure there is room for at least one more then what is actually needed.

Availability: All devices

Requires: #USE RS232

Examples:

```
#USE_RS232 (UART1,BAUD=9600,TRANSMIT_BUFFER=5
0)
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: `value = tx_buffer_bytes([stream]);`

Parameters: `stream` – optional parameter specifying the stream defined in #USE RS232.

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:

```
#USE_RS232(UART1,BAUD=9600,TRANSMIT_BUFFER=
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:	<pre>#USE_RS232(UART1,BAUD=9600,TRANSMIT_BUFFER= 50) void main(void) { char c; if(!tx_buffer_full()) putc(c); }</pre>
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:	va_arg(argptr, type)
Parameters:	<p>argptr is a special argument pointer of type va_list</p> <p>type – This is data type like int or char.</p>
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: #INCLUDE <stdarg.h>

Examples:

```
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_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:

```
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()

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:

```
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:	<pre>int data[6]; write_configuration_memory(data, 6)</pre>
Example Files:	None
Also See:	WRITE_PROGRAM_MEMORY(), Configuration Memory Overview

write_eeprom()

Syntax:	<pre>write_eeprom (address, value) write_eeprom (address , pointer , N)</pre>
Parameters:	<p>address is the 0 based starting location of the EEPROM write</p> <p>N specifies the number of EEPROM bytes to write</p> <p>value is a constant or variable to write to EEPROM</p> <p>pointer is a pointer to location to data to be written to EEPROM</p>
Returns:	undefined
Function:	<p>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.</p>
Availability:	This function is only available on devices with supporting hardware on chip.
Requires:	Nothing
Examples:	<pre>#define IAST_VOLUME 10 // Location in EEPROM volume++;</pre>

```
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 than 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:

```
for(i=0x1000;i<=0x1fff;i++) {
    value=read_adc();
    write_program_memory(i, value, 2);
    delay_ms(1000);
}

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 made in the device's header file and are as follows:

- ZCD_IS_SINKING
- ZCD_IS_SOURCING

Function: 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.

Availability: All devices with a ZCD module.

Examples: `value=zcd_status() :`

Example Files: *None*

Also See: `setup_zcd()`

STANDARD C INCLUDE FILES

errno.h

errno.h	
EDOM	Domain error value
ERANGE	Range error value
errno	error value

float.h

float.h	
FLT_RADIX:	Radix of the exponent representation
FLT_MANT_DIG:	Number of base digits in the floating point significant
FLT_DIG:	Number of decimal digits, q, such that any floating point number with q 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.
FLT_MIN_EXP:	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number.
FLT_MIN_10_EXP:	Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.
FLT_MAX_EXP:	Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number.
FLT_MAX_10_EXP:	Maximum negative integer such that 10 raised to that power is in the range representable finite floating-point numbers.
FLT_MAX:	Maximum representable finite floating point number.
FLT_EPSILON:	The difference between 1 and the least value greater than 1 that is representable in the given floating point type.
FLT_MIN:	Minimum normalized positive floating point number
DBL_MANT_DIG:	Number of base digits in the double significant
DBL_DIG:	Number of decimal digits, q, such that any double number with q decimal digits can be rounded into a double number with p radix b digits and back again without change to the q decimal digits.
DBL_MIN_EXP:	Minimum negative integer such that FLT_RADIX raised to that power 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 q 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.
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

limits.h	
CHAR_BIT:	Number of bits for the smallest object that is not a bit_field.
SCHAR_MIN:	Minimum value for an object of type signed char
SCHAR_MAX:	Maximum value for an object of type signed char
UCHAR_MAX:	Maximum value for an object of type unsigned char
CHAR_MIN:	Minimum value for an object of type char(unsigned)
CHAR_MAX:	Maximum value for an object of type char(unsigned)
MB_LEN_MAX:	Maximum number of bytes in a multibyte character.
SHRT_MIN:	Minimum value for an object of type short int
SHRT_MAX:	Maximum value for an object of type short int
USHRT_MAX:	Maximum value for an object of type unsigned short int
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)
lconv	localization structure
SETLOCALE()	returns null
LOCALCONV()	returns clocale

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

stdio.h	
stderr	The standard error s stream (USE RS232 specified as stream or the first USE RS232)
stdout	The standard output stream (USE RS232 specified as stream last USE RS232)
stdin	The standard input s stream (USE RS232 specified as stream last USE RS232)

stdlib.h

stdlib.h	
div_t	structure type that contains two signed integers (quot and rem).
ldiv_t	structure type that contains two signed longs (quot and rem)
EXIT_FAILURE	returns 1
EXIT_SUCCESS	returns 0
RAND_MAX-	
MBCUR_MAX-	1
SYSTEM()	Returns 0(not supported)
Multibyte character and string	Multibyte characters not supported
functions:	
MBLEN()	Returns the length of the string.
MBTOWC()	Returns 1.
WCTOMB()	Returns 1.
MBSTOWCS()	Returns length of string.
WBSTOMBS()	Returns length of string.

Stdlib.h functions included just for compliance with ANSI C.

SOFTWARE LICENSE AGREEMENT

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.

©1994-2015 Custom Computer Services, Inc.
ALL RIGHTS RESERVED WORLDWIDE
PO BOX 2452
BROOKFIELD, WI 53008 U.S.A.



Компания «ЭлектроПласт» предлагает заключение долгосрочных отношений при поставках импортных электронных компонентов на взаимовыгодных условиях!

Наши преимущества:

- Оперативные поставки широкого спектра электронных компонентов отечественного и импортного производства напрямую от производителей и с крупнейших мировых складов;
- Поставка более 17-ти миллионов наименований электронных компонентов;
- Поставка сложных, дефицитных, либо снятых с производства позиций;
- Оперативные сроки поставки под заказ (от 5 рабочих дней);
- Экспресс доставка в любую точку России;
- Техническая поддержка проекта, помощь в подборе аналогов, поставка прототипов;
- Система менеджмента качества сертифицирована по Международному стандарту ISO 9001;
- Лицензия ФСБ на осуществление работ с использованием сведений, составляющих государственную тайну;
- Поставка специализированных компонентов (Xilinx, Altera, Analog Devices, Intersil, Interpoint, Microsemi, Aeroflex, Peregrine, Syfer, Eurofarad, Texas Instrument, Miteq, Cobham, E2V, MA-COM, Hittite, Mini-Circuits, General Dynamics и др.);

Помимо этого, одним из направлений компании «ЭлектроПласт» является направление «Источники питания». Мы предлагаем Вам помощь Конструкторского отдела:

- Подбор оптимального решения, техническое обоснование при выборе компонента;
- Подбор аналогов;
- Консультации по применению компонента;
- Поставка образцов и прототипов;
- Техническая поддержка проекта;
- Защита от снятия компонента с производства.



Как с нами связаться

Телефон: 8 (812) 309 58 32 (многоканальный)

Факс: 8 (812) 320-02-42

Электронная почта: org@eplast1.ru

Адрес: 198099, г. Санкт-Петербург, ул. Калинина, дом 2, корпус 4, литера А.