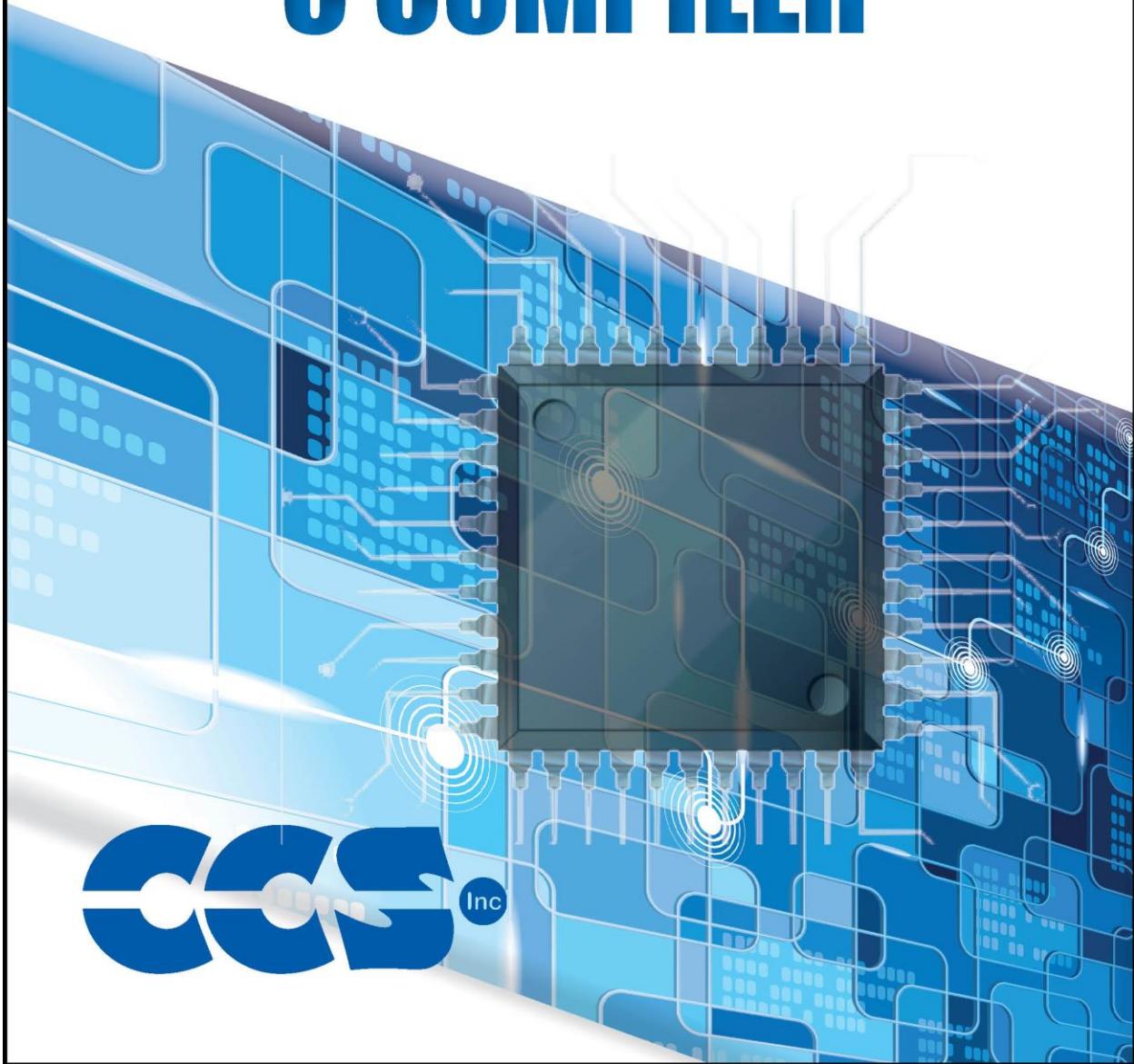


PIC® MCU C COMPILER



**PCD
June 2013**

Table of Contents

Overview.....	13
PCD	13
Technical Support	13
Directories	13
File Formats	14
Invoking the Command Line Compiler.....	15
PCW Overview	17
Program Syntax	18
Overall Structure	18
Comment.....	18
Trigraph Sequences	19
Multiple Project Files	19
Multiple Compilation Units	20
Example	20
Statements.....	22
if	22
while	22
do-while	23
for.....	23
switch	23
return.....	24
goto	24
label	24
break	25
continue.....	25
expr	25
;	26
stmt	26
Expressions	27
Operators	27
Operator Precedence	28
Reference Parameters	28
Variable Argument Lists	29
Default Parameters.....	29
Overloaded Functions	30
Data Definitions.....	31
Basic and Special types	31
Declarations	34
Non-RAM Data Definitions.....	34
Using Program Memory for Data	35
Function Definition.....	37
Functional Overview.....	38
I2C	38
ADC	39
Analog Comparator	40
CAN Bus	40
Code Profile.....	44
Configuration Memory	45
CRC	46
DAC	47
Data Eeprom	47
DCI.....	48
DMA	49
Data Signal Modulator	50

Extended RAM	51
General Purpose I/O.....	52
Input Capture.....	53
Internal Oscillator	53
Interrupts	54
Output Compare/PWM Overview.....	55
Motor Control PWM.....	56
PMP/EPMP	57
Program Eeprom.....	58
QEI.....	60
RS232 I/O	60
RTCC	62
RTOS	63
SPI	65
Timers	66
TimerA.....	67
TimerB.....	67
Voltage Reference.....	68
WDT or Watch Dog Timer	68
Pre-Processor.....	70
#ASM #ENDASM	70
#BANK_DMA.....	78
#BANKX	78
#BANKY	79
#BIT	79
#BUILD.....	80
#BYTE.....	81
#CASE	81
_DATE.....	82
#DEFINE	82
#DEFINEDINC	83
#DEVICE.....	84
_DEVICE.....	86
#ERROR	86
#EXPORT (options).....	87
__FILENAME__	88
#FILL_ROM.....	88
#FUSES	89
#HEXCOMMENT	90
#ID	90
#IF exp #ELSE #ELIF #ENDIF	91
#IFDEF #IFNDEF #ELSE #ELIF #ENDIF	92
#IGNORE_WARNINGS.....	92
#IMPORT (options).....	93
#include	94
#INLINE	95
#INT_xxxx	95
#INT_DEFAULT	99
__LINE__	100
#LIST	100
#LINE	100
#LOCATE.....	101
#MODULE	102
#NOLIST	102
#OCS	103
#OPT.....	103

#ORG	104
#PIN_SELECT	105
_ _PCD_	110
#PRAGMA	111
#PROFILE	111
#RECURSIVE	112
#RESERVE	112
#ROM	113
#SEPARATE	113
#SERIALIZE	114
# Stream I/O	116
#TASK	117
__ TIME __	118
#TYPE	119
#UNDEF	120
#USE CAPTURE	121
#USE DELAY	122
#USE DYNAMIC_MEMORY	123
#USE FAST_IO	123
#USE FIXED_IO	123
#USE I2C	124
#USE PROFILE()	125
#USE PWM	126
#USE RS232	127
#USE RTOS	130
#USE SPI	131
#USE STANDARD_IO	133
#USE TIMER	133
#USE TOUCHPAD	135
#WARNING	136
#WORD	136
#ZERO_RAM	137
Built-in Functions	138
BUILT-IN FUNCTIONS	138
abs()	145
sin() cos() tan() asin() acos() atan() sinh() cosh() tanh() atan2()	146
adc_done() adc_done2()	147
sin() cos() tan() asin() acos() atan() sinh() cosh() tanh() atan2()	148
assert()	149
sin() cos() tan() asin() acos() atan() sinh() cosh() tanh() atan2()	150
sin() cos() tan() asin() acos() atan() sinh() cosh() tanh() atan2()	151
atoe	153
atof() atof48() atof64()	153
pin_select()	154
atoi() atol() atoi32() atoi48() atoi64()	155
atoi() atol() atoi32() atoi48() atoi64()	155
atoi() atol() atoi32() atoi48() atoi64()	156
bit_clear()	157
bit_first()	158
bit_last()	158
bit_set()	159
bit_test()	159
bsearch()	160
calloc().	161
ceil()	161
clear_interrupt()	162

sin() cos() tan() asin() acos()	162
atan() sinh() cosh() tanh() atan2()	162
sin() cos() tan() asin() acos()	163
atan() sinh() cosh() tanh() atan2()	163
crc_calc() crc_calc8() crc_calc16()	165
crc_calc32()	165
crc_init(mode)	166
cwg_status()	166
cwg_restart()	167
dac_write()	167
dci_read()	168
dci_start()	168
dci_transmit_ready()	169
dci_write()	170
delay_cycles()	170
delay_ms()	171
delay_us()	172
disable_interrupts()	172
div() ldiv()	173
dma_start()	174
dma_status()	175
enable_interrupts()	175
erase_program_memory	176
exp()	177
ext_int_edge()	177
fabs()	178
getc() getch() getchar() fgetc()	178
gets() fgets()	179
floor()	180
fmod()	180
printf() fprintf()	181
putc() putchar() fputc()	182
puts() fputs()	183
free()	184
frexp()	184
get_capture()	185
get_capture_event()	185
get_capture_time()	186
get_motor_pwm_count()	186
get_nco_accumulator()	187
get_nco_inc_value()	187
get_ticks()	187
get_timerA()	188
get_timerB()	188
get_timerx()	189
get_timerxy()	189
get_tris_X()	190
getc() getch() getchar() fgetc()	191
getc() getch() getchar() fgetc()	191
getc() getch() getchar() fgetc()	192
getenv()	193
gets() fgets()	196
goto_address()	197
high_speed_adc_done()	197
i2c_init()	198
i2c_isr_state()	198
i2c_poll()	199
i2c_read()	200
i2c_slaveaddr()	200

i2c_speed()	201
i2c_start()	201
i2c_stop()	202
i2c_write()	203
input()	203
input_change_x()	204
input_state()	205
input_x()	205
interrupt_active()	206
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	206
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	207
isamong()	208
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	209
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	210
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	211
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	212
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	213
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	214
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	215
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	216
isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char)	
isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)	217
itoa()	218
kbhit()	218
label_address()	219
labs()	220
ldexp()	220
log()	221
log10()	221
longjmp()	222
make8()	222
make16()	223
make32()	223
malloc()	224
memcpy() memmove()	225
memset()	225
modf()	226
_mul()	226
nargs()	227
offsetof() offsetofbit()	228
offsetof() offsetofbit()	229
output_x()	229
output_x()	230
output_x()	231
output_x()	231
output_x()	232

output_x()	233
output_x()	233
output_x()	234
output_x()	234
output_x()	235
output_bit()	236
output_drive()	237
output_float()	237
output_high()	238
output_low()	238
output_toggle()	239
perror()	239
pmp_address(address)	240
pmp_output_full() pmp_input_full() pmp_overflow() pmp_error() pmp_timeout()	240
pmp_read()	241
pmp_write()	242
port_x_pullups()	243
pow() pwr()	244
printf() fprintf()	244
profileout()	246
psp_output_full() psp_input_full() psp_overflow()	247
psp_output_full() psp_input_full() psp_overflow()	247
psp_output_full() psp_input_full() psp_overflow()	248
psp_read()	248
psp_write()	249
putc() putchar() fputc()	250
putc_send();	250
fputc_send();	250
putc() putchar() fputc()	251
puts() fputs()	252
pwm_off()	252
pwm_on()	253
pwm_set_duty()	253
pwm_set_duty_percent	254
pwm_set_frequency	254
qei_get_count()	255
qei_set_count()	255
qei_status()	256
qsort()	256
rand()	257
rcv_buffer_bytes()	257
rcv_buffer_full()	258
read_adc() read_adc2()	258
read_configuration_memory()	259
read_eeprom()	260
read_extended_ram()	260
read_program_memory()	261
read_high_speed_adc()	261
read_program_memory()	263
read_rom_memory()	263
realloc()	264
reset_cpu()	264
restart_cause()	265
restart_wdt()	265
rotate_left()	266
rotate_right()	267

rtc_alarm_read()	267
rtc_alarm_write()	268
rtc_read()	268
rtc_write()	269
rtos_await()	269
rtos_disable()	270
rtos_enable()	270
rtos_msg_poll()	271
rtos_msg_read()	271
rtos_msg_send()	271
rtos_overrun()	272
rtos_run()	272
rtos_signal()	273
rtos_stats()	273
rtos_terminate()	274
rtos_wait()	274
rtos_yield()	275
set_adc_channel()	275
set_adc_channel2()	275
set_compare_time()	276
set_motor_pwm_duty()	276
set_motor_pwm_event()	277
set_motor_unit()	277
set_nco_inc_value()	278
set_pullup()	279
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	279
set_ticks()	280
set_timerA()	281
set_timerB()	281
set_timerx()	281
set_timerxy()	282
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	282
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	283
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	284
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	285
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	285
set_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()	286
set_tris_x()	287
set_tris_X()	287
set_tris_x()	288
set_tris_x()	289
set_tris_X()	290
set_tris_x()	290
set_tris_X()	291
set_tris_x()	292
set_tris_X()	292
set_tris_x()	293
set_uart_speed()	294
setjmp()	294

setup_adc(mode)	295
setup_adc2(mode).....	295
setup_adc_ports().....	295
setup_adc_ports2().....	295
setup_capture()	296
setup_clc1() setup_clc2() setup_clc3() setup_clc4()	297
setup_comparator()	297
setup_compare()	298
setup_crc(mode)	299
setup_cwg().....	299
setup_dac()	300
setup_dci()	301
setup_dma()	302
setup_high_speed_adc()	302
setup_high_speed_adc_pair()	303
setup_low_volt_detect().....	303
setup_motor_pwm().....	304
setup_oscillator()	305
setup_pmp(option,address_mask)	305
setup_power_pwm_pins().....	306
setup_psp(option,address_mask).	307
setup_pwm1() setup_pwm2() setup_pwm3() setup_pwm4()	308
setup_qei()	308
setup_rtc()	309
setup_rtc_alarm()	309
setup_spi() setup_spi2()	310
setup_timerx()	310
setup_timer_A()	312
setup_timer_B()	312
setup_timer_0()	313
setup_timer_1()	313
setup_timer_2()	314
setup_timer_3()	314
setup_timer_4()	315
setup_timer_5()	315
setup_uart().....	316
setup_vref()	317
setup_wdt().....	317
shift_left()	318
shift_right()	318
sin() cos() tan() asin() acos() atan() sinh() cosh() tanh() atan2()	319
sin() cos() tan() asin() acos() atan() sinh() cosh() tanh() atan2()	321
sleep()	322
spi_data_is_in() spi_data_is_in2()	323
spi_init()	323
spi_prewrite(data);.....	324
spi_read() spi_read2()	324
spi_read3()	324
spi_read4()	324
spi_read_16()	325
spi_read2_16()	325
spi_read3_16()	325
spi_read4_16()	325
spi_speed	325
spi_write() spi_write2()	326
spi_write3()	326

strtol()	350
strtoul()	350
STANDARD STRING FUNCTIONS() memchr() memcmp() strcat() strchr() strcmp()	
strcoll() strcspn() strerror() strcmp() strlen() strlwr() strncat() strncmp() strncpy()	
strpbrk() strrchr() strspn() strstr() strxfrm()	351
swap()	352
sin() cos() tan() asin() acos() atan() sinh() cosh() tanh() atan2()	353
sin() cos() tan() asin() acos() atan() sinh() cosh() tanh() atan2()	354
tolower() toupper()	356
touchpad_getc()	356
touchpad_hit()	357
touchpad_state()	357
tx_buffer_bytes()	358
tx_buffer_full()	359
va_arg()	359
va_end()	360
va_start	360
write_configuration_memory()	361
write_eeprom()	361
write_extended_ram().....	362
write_program_memory()	362
Standard C Include Files	364
errno.h.....	364
float.h	364
limits.h.....	365
locale.h.....	366
setjmp.h.....	366
stddef.h	366
stdio.h.....	366
stdlib.h.....	366
Error Messages.....	368
Compiler Error Messages	368
Compiler Warning Messages	376
Compiler Warning Messages	376
Common Questions & Answers	379
How are type conversions handled?	379
How can a constant data table be placed in ROM?	380
How can I use two or more RS-232 ports on one PIC®?	380
How do I directly read/write to internal registers?	381
How do I do a printf to a string?	382
How do I get getc() to timeout after a specified time?	382
How do I make a pointer to a function?.....	383
How do I wait only a specified time for a button press?	383
How do I write variables to EEPROM that are not a word?	383
How does one map a variable to an I/O port?	384
How does the compiler determine TRUE and FALSE on expressions?	385
How does the PIC® connect to a PC?	385
How does the PIC® connect to an I2C device?	386
How much time do math operations take?	386
What can be done about an OUT OF RAM error?	386
What is an easy way for two or more PICs® to communicate?	387
What is an easy way for two or more PICs® to communicate?	387
What is the format of floating point numbers?.....	387
Why does the .LST file look out of order?	388
Why is the RS-232 not working right?.....	389
Example Programs	391

EXAMPLE PROGRAMS.....	391
Software License Agreement	401
SOFTWARE LICENSE AGREEMENT.....	401
Index.....	402

OVERVIEW

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 CORCON
	Symbols	Shows variable names instead of addresses
	Interpret	Adds a pseudo code interpretation to the right of assembly instruction understand the operation. For example: <code>LSR W4, #8, W5 : W5=W4>>8</code>

.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.

.hex	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
d	
.ccssio	used to link Windows 8 apps to Serial Port Monitor
w	

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	Do not create 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 SXC (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.

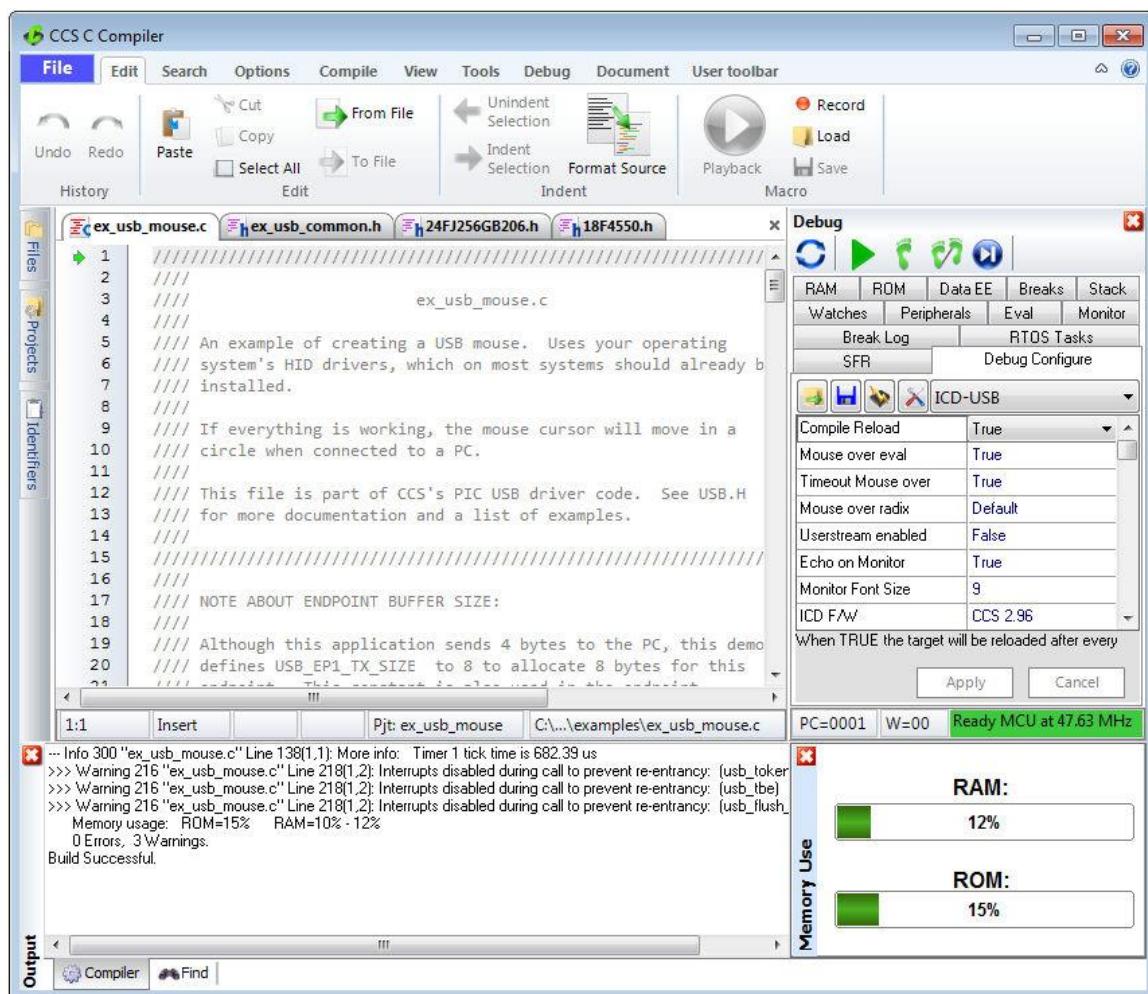
Examples:

```
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. Click on an image or title for more help on that particular component.

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



For details and additional information on all PCW IDE features, please use the Help File in the compiler.

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

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:

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

Traditionally, the CCS C compiler used only one compilation unit and multiple files were implemented with #include files. 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.

Example

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
#fuses NOPROTECT              // Code protection turned off
```

```

#define delay(crystal=20mhz) // preprocessor directive that specifies the clock
type and speed
#define 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:");
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

if

if-else

The if-else statement is used to make decisions.

The syntax is:

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

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

else-if

This is used to make multi-way decisions.

The syntax is:

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

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

Example:

```
if (x==25)
    x=1;
else
    x=x+1;
```

Also See: Statements

while

While is used as a loop/iteration statement.

The syntax is:

```
while (expr)
    statement
```

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

Example:

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

Also See: Statements

do-while

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

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

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

Also See: Statements , [While](#)

for

For is also used as a loop/iteration statement.

The syntax is:

```
for (expr1;expr2;expr3)
    statement
```

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

Example:

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

Also See: Statements

switch

Switch is also a special multi-way decision maker.

The syntax is

```
switch (expr) {
    case const1: stmt sequence;
        break;
    ...
    [default:stmt]
}
```

This tests whether the expression matches one of the constant values and branches accordingly.

If none of the cases are satisfied the default case is executed. The break causes an immediate exit, otherwise control falls through to the next case.

Example:

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

Also See: Statements

return

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

Operators

+	Addition Operator
$+=$	Addition assignment operator, $x+=y$, is the same as $x=x+y$
$\&=$	Bitwise and assignment operator, $x\&=y$, is the same as $x=x\&y$
&	Address operator
&	Bitwise and operator
$\wedge=$	Bitwise exclusive or assignment operator, $x\wedge=y$, is the same as $x=x\wedge y$
\wedge	Bitwise exclusive or operator
$ =$	Bitwise inclusive or assignment operator, $x =y$, is the same as $x=x y$
	Bitwise inclusive or operator
?:	Conditional Expression operator
--	Decrement
$/=$	Division assignment operator, $x/=y$, is the same as $x=x/y$
/	Division operator
$==$	Equality
>	Greater than operator
\geq	Greater than or equal to operator
$++$	Increment
*	Indirection operator
$!=$	Inequality
$<=$	Left shift assignment operator, $x<<=y$, is the same as $x=x<<y$
<	Less than operator
$<<$	Left Shift operator
\leq	Less than or equal to operator
$\&\&$	Logical AND operator
!	Logical negation operator
$\ $	Logical OR operator
$\%=$	Modules assignment operator $x\%-=y$, is the same as $x=x\%y$
%	Modules operator
$*=$	Multiplication assignment operator, $x*=y$, is the same as $x=x*y$
*	Multiplication operator
-	One's complement operator
$>>=$	Right shift assignment, $x>>=y$, is the same as $x=x>>y$
>>	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

Operator Precedence

PIN DESCENDING PRECEDENCE

(expr)			
++expr	expr++	- -expr	expr - -
!expr	~expr	+expr	-expr
(type)expr	*expr	&value	sizeof(type)
expr*expr	expr/expr	expr%expr	
expr+expr	expr-expr		
expr<<expr	expr>>expr		
expr<expr	expr<=expr	expr>expr	expr>=expr
expr==expr	expr!=expr		
expr&expr			
expr^expr			
expr expr			
expr&& expr			
expr expr			
expr ? expr: expr			
lvalue = expr	lvalue+=expr	lvalue-=expr	
lvalue*=expr	lvalue/=expr	lvalue%=expr	
lvalue>>expr	lvalue<<expr	lvalue&=expr	
lvalue^=expr	lvalue =expr		
expr, expr			

(Operators on the same line are equal in precedence)

Reference Parameters

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

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

```
funct_b(int&x,int&y){  
    /*Reference params*/  
    if(x!=5)  
        y=x+3;
```

```
}

funct_b(a,b);
```

Variable Argument Lists

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

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

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

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

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

Some examples of using this new function:

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

Default Parameters

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

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

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

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

Overloaded Functions

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

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

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

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

This function finds the square root of a float variable.

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

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

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

DATA DEFINITIONS

Basic and Special types

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.

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	-1.5×10^{45} to 3.4×10^{38}		7-8
float48	48 bit float (higher precision)	-2.9×10^{39} to 1.7×10^{38}		11-12
float64	64 bit float	-5.0×10^{324} to 1.7×10^{308}		15-16

C Standard Type

C Standard Type	Default Type
short	unsigned int8
char	unsigned int8
int	unsigned int16
long	unsigned int32
long long	unsigned int64
float	float32
double	

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

Special 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
```

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 a even byte boundary. Some Bits may by 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_tab {
    int ival;
    long lval;
    float fval;
};           // u_tag is a union type that can hold a float
```

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, green=2,blue}colors; //colors can be used to
declare
                                //variables of this enum type

```

__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

```

Declarations

A declaration specifies 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

```

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 width 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.

Function Definition

The format of a function definition is as follows:

[qualifier] id	([type-specifier id])	{ [stmt] }
Optional See Below	Zero or more comma separated. See Data Types	Zero or more Semi-colon separated. See Statements.

The qualifiers for a function are as follows:

- VOID
- type-specifier
- #separate
- #inline
- #int_..

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

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

Example:

```
void lcd_putc(char c) {  
    ...  
}  
  
lcd_putc ("Hi There.");
```

FUNCTIONAL OVERVIEW

I2C

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

Relevant Functions:

<u>i2c_start()</u>	Issues a start command when in the I2C master mode.
<u>i2c_write(data)</u>	Sends a single byte over the I2C interface.
<u>i2c_read()</u>	Reads a byte over the I2C interface.
<u>i2c_stop()</u>	Issues a stop command when in the I2C master mode.
<u>i2c_poll()</u>	Returns a TRUE if the hardware has received a byte in the buffer.

Relevant Preprocessor:

<u>#USE I2C</u>	Configures the compiler to support I2C™ to your specifications.
---------------------------------	---

Relevant Interrupts:

#INT_SSP	I2C or SPI activity
#INT_BUSCOL	Bus Collision
#INT_I2C	I2C Interrupt (Only on 14000)
#INT_BUSCOL2	Bus Collision (Only supported on some PIC18's)
#INT_SSP2	I2C or SPI activity (Only supported on some PIC18's)
#INT_mi2c	Interrupts on activity from the master I2C module
#INT_si2c	Interrupts on activity from the slave I2C module

Relevant Include Files:

None, all functions built-in

Relevant getenv() Parameters:

I2C_SLAVE	Returns a 1 if the device has I2C slave H/W
I2C_MASTER	Returns a 1 if the device has a I2C master H/W

Example Code:

```
#define Device_SDA PIN_C3 // Pin defines
#define Device_SLC PIN_C4
use i2c(master,
sda=Device_SDA,
scl=Device_SCL)
..
BYT data; // Data to be transmitted
i2c_start(); // Issues a start command when in the I2C master mode.
i2c_write(data); // Sends a single byte over the I2C interface.
i2c_stop(); // Issues a stop command when in the I2C master mode.
```

ADC

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

Relevant Functions:

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

Relevant Preprocessor:

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

Relevant Interrupts:

<code>INT_AD</code>	Interrupt fires when a/d conversion is complete
<code>INT_ADOF</code>	Interrupt fires when a/d conversion has timed out

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

<code>ADC_CHANNELS</code>	Number of A/D channels
<code>ADC_RESOLUTION</code>	Number of bits returned by <code>read_adc</code>

Example Code:

```
#DEVICE ADC=10
...
long value;
...
setup_adc(ADC_CLOCK_INTERNAL); //enables the a/d module
                                //and sets the clock to internal adc clock
setup_adc_ports(ALL_ANALOG);   //sets all the adc pins to analog
set_adc_channel(0);           //the next read_adc call will read channel 0
delay_us(10);                //a small delay is required after setting the channel
                                //and before read
```

```

value=read_adc();           //starts the conversion and reads the result
                           //and store it in value
read_adc(ADC_START_ONLY); //only starts the conversion
value=read_adc(ADC_READ_ONLY); //reads the result of the last conversion and store it in //value. Assuming
                               //device has a 10bit ADC module, //value will range between 0-3FF. If #DEVICE
                               //ADC=8 had //been used instead the result will yield 0-FF. If #DEVICE
                               //ADC=16 had been used instead the result will yield 0-//FFC0

```

Analog Comparator

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

Relevant Functions:

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

Relevant Preprocessor:

None

Relevant Interrupts:

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

Relevant Include Files:

None, all functions built-in

Relevant getenv() Parameters:

Returns 1 if the device has a comparator	COMP
--	------

Example Code:

```

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

```

CAN Bus

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

Relevant Functions:

can_init(void);	Initializes the CAN module and clears all the filters and masks so that all messages can be received from any ID.
can_set_baud(void);	Initializes the baud rate of the CAN bus to 125kHz, if

using a 20 MHz clock and the default CAN-BRG defines, it is called inside the can_init() function so there is no need to call it.

can_set_mode (CAN_OP_MODE mode);	Allows the mode of the CAN module to be changed to configuration mode, listen mode, loop back mode, disabled mode, or normal mode.
can_set_functional_mode (CAN_FUN_OP_MODE mode);	Allows the functional mode of ECAN modules to be changed to legacy mode, enhanced legacy mode, or first in firstout (fifo) mode. ECAN
can_set_id(int* addr, int32 id, int1 ext);	Can be used to set the filter and mask ID's to the value specified by addr. It is also used to set the ID of the message to be sent.
can_get_id(int * addr, int1 ext);	Returns the ID of a received message.
can_putd (int32 id, int * data, int len, int priority, int1 ext, int1 rtr);	Constructs a CAN packet using the given arguments and places it in one of the available transmit buffers.
can_getd (int32 & id, int * data, int & len, struct rx_stat & stat);	Retrieves a received message from one of the CAN buffers and stores the relevant data in the referenced function parameters.
can_enable_rtr(PROG_BUFFER b);	Enables the automatic response feature which automatically sends a user created packet when a specified ID is received. ECAN
can_disable_rtr(PROG_BUFFER b);	Disables the automatic response feature. ECAN
can_load_rtr (PROG_BUFFER b, int * data, int len);	Creates and loads the packet that will automatically transmitted when the triggering ID is received. ECAN
can_enable_filter(long filter);	Enables one of the extra filters included in the ECAN module. ECAN
can_disable_filter(long filter);	Disables one of the extra filters included in the ECAN module. ECAN
can_associate_filter_to_buffer (CAN_FILTER_ASSOCIATION_BUFFERS buffer,CAN_FILTER_ASSOCIATION filter);	Used to associate a filter to a specific buffer. This allows only specific buffers to be filtered and is available in the ECAN module. ECAN
can_associate_filter_to_mask (CAN_MASK_FILTER_ASSOCIATE mask, CAN_FILTER_ASSOCIATION filter);	Used to associate a mask to a specific buffer. This allows only specific buffer to have this mask applied. This feature is available in the ECAN module. ECAN
can_fifo_getd(int32 & id,int * data, int &len,struct rx_stat & stat);	Retrieves the next buffer in the fifo buffer. Only available in the ECON module while operating in fifo mode. ECAN

Relevant Preprocessor:

None

Relevant Interrupts:

#int_canirx	This interrupt is triggered when an invalid packet is received on the CAN.
#int_canwake	This interrupt is triggered when the PIC is woken up by activity on the CAN.
#int_canerr	This interrupt is triggered when there is an error in the CAN module.
#int_cantx0	This interrupt is triggered when transmission from buffer 0 has completed.
#int_cantx1	This interrupt is triggered when transmission from buffer 1 has completed.
#int_cantx2	This interrupt is triggered when transmission from buffer 2 has completed.
#int_canrx0	This interrupt is triggered when a message is received in buffer 0.
#int_canrx1	This interrupt is triggered when a message is received in buffer 1.

Relevant Include Files:

can-mcp2510.c	Drivers for the MCP2510 and MCP2515 interface chips
can-18xxx8.c	Drivers for the built in CAN module
can-18F4580.c	Drivers for the build in ECAN module

Relevant getenv() Parameters:

none

Example Code:

```
can_init(); // initializes the CAN bus
can_putd(0x300,data,8,3,TRUE, FALSE); // places a message on the CAN buss with
// ID = 0x300 and eight bytes of data pointed to by
// "data", the TRUE creates an extended ID, the
// FALSE creates
can_getd(ID,data,len,stat); // retrieves a message from the CAN bus storing the
// ID in the ID variable, the data at the array pointed
// to by
// "data", the number of data bytes in len, and
// statistics
// about the data in the stat structure.
```

Relevant Functions:

can_init(void);	Initializes the module to 62.5k baud for ECAN and 125k baud for CAN and clears all the filters and masks so that all messages can be received from any ID.
can_set_baud(void);	Initializes the baud rate of the bus to 62.5kHz for ECAN and 125kHz for CAN. It is called inside the can_init() function so there is no need to call it again.
can_set_mode (CAN_OP_MODE mode);	Allows the mode of the CAN module to be changed to listen all mode configuration mode, listen mode, loop back mode, disabled mode, or

	normal mode.
can_set_functional_mode (CAN_FUN_OP_MODE mode);	Allows the functional mode of ECAN modules to be changed to legacy mode, enhanced legacy mode, or first in firstout (fifo) mode. ECAN
can_set_id(int16 *addr, int32 id, int1 ext)	Can be used to set the filter and mask ID's to the value specified by the arguments. It is also used to set the ID of the message to be sent on CAN chips.
can_set_buffer_id(BUFFER buffer, int32 id, int1 ext)	Can be used to set the ID of the message to be sent for ECAN chips.
can_get_id(BUFFER buffer, int1 ext)	Returns the ID of a received message.
can_putd(int32 id, int8 *data, int8 len, int8 priority, int1 ext, int1 rtr)	Constructs a CAN packet using the given arguments and places it in one of the available transmit buffers.
can_getd(int32 &id, int8 *data, int8 &len, struct rx_stat &stat)	Retrieves a received message from one of the CAN buffers and stores relevant data in the referenced function parameters.
can_kbhit()	Returns TRUE if valid CAN messages are available to be retrieved from the receive buffers.
can_tbe()	Returns TRUE if a transmit buffer is available to send more data.
can_abort()	Aborts all pending transmissions.
can_enable_b_transfer(BUFFER b)	Sets the specified programmable buffer to be a transmit buffer. ECAN
can_enable_b_receiver(BUFFER b)	Sets the specified programmable buffer to be a receive buffer. By default all programmable buffers are set to be receive buffers. ECAN
can_enable_rtr(BUFFER b)	Enables the automatic response feature which automatically sends a user created packet when a specified ID is received. ECAN
can_disable_rtr(BUFFER b)	Disables the automatic response feature. ECAN
can_load_rtr (BUFFER b, int8 *data, int8 len)	Creates and loads the packet that will automatically transmitted when the triggered ID is received. ECAN
can_set_buffer_size(int8 size)	Set the number of buffers to use. Size can be 4, 6, 8, 12, 16, 24, and 32. By default can_init() sets size to 32. ECAN
can_enable_filter (CAN_FILTER_CONTROL filter)	Enables one of the acceptance filters included in the ECAN module. ECAN
can_disable_filter (CAN_FILTER_CONTROL filter)	Disables one of the acceptance filters included in the ECAN module. ECAN
can_associate_filter_to_buffer (CAN_FILTER_ASSOCIATION_BUFFERS buffer, CAN_FILTER_ASSOCIATION filter)	Used to associate a filter to a specific buffer. This allows only specific buffers to be filtered and is available in the ECAN module. ECAN
can_associate_filter_to_mask (CAN_MASK_FILTER_ASSOCIATION mask, CAN_FILTER_ASSOCIATION filter)	Used to associate a mask to a specific buffer. This allows only specific buffers to have this mask applied. This feature is available in the ECAN module. ECAN
can_fifo_getd(int32 &id, int8 *data, int8 &len, struct rx_stat &stat)	Retrieves the next buffer in the FIFO buffer. Only available in the ECAN module. ECAN
can_trb0_putd(int32 id, int8 *data, int8 len, int8 pri, int1 ext, int1 rtr)	Constructs a CAN packet using the given arguments and places it in transmit buffer 0. Similar functions available for all transmit buffers 0-7. Buffer must be made a transmit buffer with can_enable_b_transfer() function before function can be used. ECAN
can_enable_interrupts(INTERRUPT setting)	Enables specified interrupt conditions that cause the #INT_CAN1 interrupt to be triggered. Available options are:
TB - Transmitt Buffer Interrupt	ECAN
RB - Receive Buffer Interrupt	ECAN
RXOV - Receive Buffer Overflow Interrupt	ECAN
FIFO - FIFO Almost Full Interrupt	ECAN
ERR - Error interrupt	ECAN/CAN
WAK - Wake-Up Interrupt	ECAN/CAN
IVR - Invalid Message Received Interrupt	ECAN/CAN
RX0 - Receive Buffer 0 Interrupt	CAN

	RX1 - Receive Buffer 1 Interrupt TX0 - Transmit Buffer 0 Interrupt TX1 - Transmit Buffer 1 Interrupt TX2 - Transmit Buffer 2 Interrupt	CAN CAN CAN CAN
can_disable_interrupts(INTERRUPT setting)	Disable specified interrupt conditions so they doesn't cause the #INT_CAN1 interrupt to be triggered. Available options are the same as for the can_enable_interrupts() function. By default all conditions are disabled.	
can_config_DMA(void)	Configures the DMA buffers to use with the ECAN module. It is called inside the can_init() function so there is no need to call it. ECAN	
For PICs that have two CAN or ECAN modules all the above function are available for the second module, and they start with can2 instead of can.	Examples: can2_init(); can2_kbhit();	
Relevant Preprocessor:		
None		
Relevant Interrupts:		
#INT_CAN1	Interrupt for CAN or ECAN module 1. This interrupt is triggered when one of the conditions set by the can_enable_interrupts() is meet.	
#INT_CAN2	Interrupt for CAN or ECAN module 2. This interrupt is triggered when one of the conditions set by the can2_enable_interrupts() is meet. This interrupt is only available on PICs that have two CAN or ECAN modules.	
Relevant Include Files:		
can-mcp2510.c	Drivers for the MCP2510 and MCP2515 interface chips.	
can-dsPIC30.c	Drivers for the built in CAN module on dsPIC30F chips.	
can-PIC24.c	Drivers for the build in ECAN module on PIC24HJ and dsPIC33FJ chips.	
Relevant getenv() Parameters:		
None		
Example Code:		
can_init();	// Initializes the CAN bus.	
can_putd(0x300,data,8,3,TRUE,FALSE);	// Places a message on the CAN bus with // ID = 0x300 and eight bytes of data pointed to by // "data", the TRUE causes an extended ID to be // sent, the FALSE causes no remote transmission // to be requested.	
can_getd(ID,data,len,stat);	// Retrieves a message from the CAN bus storing the // ID in the ID variable, the data at the array // pointed to by // to by "data", the number of data bytes in len and staticstics // / about the data in the stat structure.	

Code Profile

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

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

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

Relevant Functions:

<code>profileout()</code>	Send a user specified message or variable to be displayed or logged by the code profile tool.
---------------------------	---

Relevant Pre-Processor:

<code>#use profile()</code>	Global configuration of the code profile run-time on the microcontroller.
-----------------------------	---

<code>#profile</code>	Dynamically enable/disable specific elements of the profiler.
-----------------------	---

Relevant Interrupts:	The profiler can be configured to use a microcontroller's internal timer for more accurate timing of events over the clock on the PC. This timer is configured using the <code>#profile</code> pre-processor command.
-----------------------------	---

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

Relevant getenv():	None
---------------------------	-------------

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

Example Code:

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

Configuration Memory

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

Relevant Functions:

<u>write_configuration_memory</u>	Writes n bytes to configuration from ramPtr, no erase needed <code>(ramPtr, n);</code>
---	---

or

<u>write_configuration_memory</u>	Write n bytes to configuration memory, starting at offset, from <code>(offset, ramPtr, n);</code>
---	--

<u>read_configuration_memory</u>	Read n bytes of configuration memory, save to ramPtr <code>(ramPtr, n);</code>
--	---

Relevant Preprocessor:	The initial value of the configuration memory is set through a <code>#FUSE</code>
-------------------------------	--

Relevant Interrupts : None

Relevant Include Files: None, all functions built-in

Relevant getenv() parameters: None

Example Code:

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

CRC

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

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

Relevant Functions:

[setup_crc \(polynomial\)](#) This will setup the CRC polynomial.

[crc_init \(data\)](#) Sets the initial value used by the CRC module.

[crc_calc \(data\)](#) Returns the calculated CRC value.

Relevant Preprocessor:

None

Relevant Interrupts :

#INT_CRC On completion of CRC calculation.

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

None

Example Code:

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

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

```
crc_init(0xFFFF); Starts the CRC accumulator out at 0xFFFF
```

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

DAC

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

Relevant Functions:

[setup_dac\(divisor\)](#) Sets up the DAC e.g. Reference voltages

[dac_write\(value\)](#) Writes the 8-bit value to the DAC module

[setup_dac\(mode, divisor\)](#) Sets up the d/a mode e.g. Right enable, clock divisor

[dac_write\(channel, value\)](#) Writes the 16-bit value to the specified channel

Relevant Preprocessor:

[#USE DELAY](#) Must add an auxiliary clock in the #use delay preprocessor. For example:

#USE DELAY(clock=20M, Aux: crystal=6M, clock=3M)

Relevant Interrupts:

None

Relevant Include Files:

None, all functions built-in

Relevant getenv()

parameters:

None

Example Code:

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

Data Eeprom

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

Relevant Functions:

(8 bit or 16 bit depending on the device)

[read_eeprom\(address\)](#) Reads the data EEPROM memory location

write_eeprom(address, value)	Erases and writes value to data EEPROM location address.
read_eeprom(address, [N])	Reads N bytes of data EEPROM starting at memory location address. The maximum return size is int64.
read_eeprom(address, [variable])	Reads from EEPROM to fill variable starting at address
read_eeprom(address, pointer, N)	Reads N bytes, starting at address, to pointer
write_eeprom(address, value)	Writes value to EEPROM address
write_eeprom(address, pointer, N)	Writes N bytes to address from pointer
Relevant Preprocessor:	
#ROM address={list}	Can also be used to put data EEPROM memory data into the hex file.
<code>write_eeprom = noint</code>	Allows interrupts to occur while the write_eeprom() operations is polling the done bit to check if the write operations has completed. Can be used as long as no EEPROM operations are performed during an ISR.
Relevant Interrupts:	
<code>INT_EEPROM</code>	Interrupt fires when EEPROM write is complete
Relevant Include Files:	
None, all functions built-in	
Relevant getenv() parameters:	
<code>DATA_EEPROM</code>	Size of data EEPROM memory.
Example Code:	

```
#ROM 0x007FFC00={1,2,3,4,5} // Inserts this data into the hex file
                                // The data EEPROM address differs between PICs
                                // Please refer to the device editor for device specific values.
write_eeprom(0x10, 0x1337);    // Writes 0x1337 to data EEPROM location 10.
value=read_eeprom(0x0);        // Reads data EEPROM location 10 returns 0x1337.
```

DCI

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

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

Relevant Preprocessor:

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

Relevant Interrupts:

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

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

None

Example Code:

```
signed int16 left_channel, right_channel;
```

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

...
dci_start();

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

DMA

The Direct Memory Access (DMA) controller facilitates the transfer of data between the CPU and its peripherals without the CPU's assistance. The transfer takes place between peripheral data registers and data space RAM. The module has 8 channels and since each channel is unidirectional, two channels must be allocated to read and write to a peripheral. Each DMA channel can move a block of up to 1024 data elements after it generates an interrupt to the CPU to indicate that the lock is available for processing. Some of the key features of the DMA module are:

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

Relevant Functions:

<code>setup_dma(channel, peripheral, mode)</code>	Configures the DMA module to copy data from the specified peripheral to RAM allocated for the DMA channel.
<code>dma_start(channel, mode, address)</code>	Starts the DMA transfer for the specified channel in the specified mode of operation.
<code>dma_status(channel)</code>	This function will return the status of the specified channel in the DMA module.

Relevant Preprocessor:

None

Relevant Interrupts :

<code>#INT_DMAX</code>	Interrupt on channel X after DMA block or half block transfer.
------------------------	--

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

None

Example Code:

```
setup_dma(1,DMA_IN_SPI1,DMA_BYTE); Setup channel 1 of the DMA module to read the  
SPI1 channel in byte mode.
```

<code>dma_start(1, DMA_CONTINOUS </code>	Start the DMA channel with the DMA RAM address of 0x2000
<code>DMA_PING_PONG, 0x2000);</code>	

Data Signal Modulator

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

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

Relevant Functions:

(8 bit or 16 bit depending on the device)

`setup_dsm(mode,source,carrier)` Configures the DSM module and selects the source signal and carrier signals.

<code>setup_dsm(TRUE)</code>	Enables the DSM module.
------------------------------	-------------------------

`setup_dsm(FALSE)` Disables the DSM module.

Relevant Preprocessor:

None

Relevant Interrupts:

None

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters: None

Example Code:

```
setup_dsm(DSM_ENABLED |           //Enables DSM module with the output enabled and selects
           DSM_OUTPUT_ENABLED,        //UART1
           DSM_SOURCE_UART1,          //as the source signal and VSS as the high carrier signal and OC1's
           DSM_CARRIER_HIGH_VSS |    //PWM output as the low carrier signal.
           DSM_CARRIER_LOW_OC1);
```



```
if(input(PIN_B0))                Disable DSM module
  setup_dsm(FALSE);
else                           Enable DSM module
  setup_dsm(TRUE);
```

Extended RAM

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

Relevant Functions:

<u>write_extended_ram(p,addr,ptr,n);</u>	Writes n bytes from ptr to extended RAM page p starting at address addr.
<u>read_extended_ram(p,addr,ptr,n);</u>	Reads n bytes from extended RAM page p starting a address addr to ptr.

Relevant Preprocessor:

None

Relevant Interrupts :

None

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

None

Example Code:

```

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

```

General Purpose I/O

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

Relevant Functions:

<u>output_high(pin)</u>	Sets the given pin to high state.
<u>output_low(pin)</u>	Sets the given pin to the ground state.
<u>output_float(pin)</u>	Sets the specified pin to the output mode. This will allow the pin to float high to represent a high on an open collector type of connection.
<u>output_x(value)</u>	Outputs an entire byte to the port.
<u>output_bit(pin,value)</u>	Outputs the specified value (0,1) to the specified I/O pin.
<u>input(pin)</u>	The function returns the state of the indicated pin.
<u>input_state(pin)</u>	This function reads the level of a pin without changing the direction of the pin as INPUT() does.
<u>set_tris_x(value)</u>	Sets the value of the I/O port direction register. A '1' is an input and '0' is for output.
<u>input_change_x()</u>	This function reads the levels of the pins on the port, and compares them to the last time they were read to see if there was a change, 1 if there was, 0 if there wasn't.

Relevant Preprocessor:

<u>#USE STANDARD_IO(port)</u>	This compiler will use this directive by default and it will automatically inserts code for the direction register whenever an I/O function like output_high() or input() is used.
<u>#USE FAST_IO(port)</u>	This directive will configure the I/O port to use the fast method of performing I/O. The user will be responsible for setting the port direction register using the set_tris_x() function.
<u>#USE FIXED_IO (port_outputs=:in,pin?)</u>	This directive sets particular pins to be used as input or output, and the compiler will perform this setup every time this pin is used.

Relevant Interrupts:

None

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

PIN:pb	Returns a 1 if bit b on port p is on this part
--------	--

Example Code:

```

#use fast io(b)
...
Int8 Tris_value= 0x0F;

```

```

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

```

Input Capture

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

Relevant Functions:

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

Relevant Preprocessor: None

Relevant Interrupts:

INT_ICx	Interrupt fires on capture event as configured
---------	--

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

Relevant getenv() parameters: None

Example Code:

```

setup timer3(TMR INTERNAL | TMR DIV BY 8);
setup capture(2, CAPTURE FE | CAPTURE TIMER3);
while(TRUE) {
    timerValue = get_capture(2, TRUE);
    printf("A module 2 capture event occurred at: %LU", timerValue);
}

```

Internal Oscillator

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

Relevant Functions:

<u>setup_oscillator()</u>	Explicitly configures the oscillator.
---	---------------------------------------

Relevant Preprocessor:	Specifies the values loaded in the device configuration memory.
#FUSES	May be used to setup the oscillator configuration.
Relevant Interrupts:	
#int_oscfail	Interrupts on oscillator failure
Relevant Include Files:	None, all functions built-in
Relevant getenv() parameters:	
CLOCK	Returns the clock speed specified by #use delay()
FUSE_SETxxxx	Returns 1 if the fuse xxxx is set.
Example Code:	None

Interrupts

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

Relevant Functions:

disable_interrupts()	Disables the specified interrupt.
enable_interrupts()	Enables the specified interrupt.
ext_int_edge()	Enables the edge on which the edge interrupt should trigger. This can be either rising or falling edge.
clear_interrupt()	This function will clear the specified interrupt flag. This can be used if a global isr is used, or to prevent an interrupt from being serviced.
interrupt_active()	This function checks the interrupt flag of specified interrupt and returns true if flag is set.

Relevant Preprocessor:

	This directive tells the compiler to generate code for high priority interrupts.
	This directive tells the compiler that the specified interrupt should be treated as a high priority interrupt.
#INT_XXX level=x	x is an int 0-7, that selects the interrupt priority level for that interrupt.
#INT_XXX fast	This directive makes use of shadow registers for fast register save.
	This directive can only be used in one ISR

Relevant Interrupts:

#int_default	This directive specifies that the following function should be called if an interrupt is triggered but no routine is associated with that interrupt.
-------------------------------------	--

#int_global This directive specifies that the following function should be called whenever an interrupt is triggered. This function will replace the compiler generated interrupt dispatcher.

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

Relevant Include Files:
none, all functions built in.

Relevant getenv()

Parameters:

none

Example Code:

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

Output Compare/PWM Overview

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

Relevant Functions:

[setup_comparex \(x, mode\)](#) Sets the *mode* of the output compare / PWM module x

[set_comparex_time \(x, ocr, \[ocrs\] \)](#) Sets the OCR and optionally OCRS register values of module x.

[set_pwm_duty \(x, value\)](#) Sets the PWM duty cycle of module x to the specified *value*

Relevant Preprocessor:

None

Relevant Interrupts:

INT_OCx Interrupt fires after a compare event has occurred

Relevant Include Files:

None, all functions built-in.

Relevant getenv() parameters:

None

Example Code:

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

Motor Control PWM

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

Relevant Functions:

<u>setup_motor_pwm(pwm,options,timebase);</u>	Configures the motor control PWM module.
<u>set_motor_pwm_duty(pwm,unit,time)</u>	Configures the motor control PWM unit duty.
<u>set_motor_pwm_event(pwm,time)</u>	Configures the PWM event on the motor control unit.
<u>set_motor_unit(pwm,unit,options,active_deadtime,inactive_deadtime);</u>	Configures the motor control PWM unit.
<u>get_motor_pwm_event(pwm);</u>	Returns the PWM event on the motor control unit.

Relevant Preprocessor:

None

Relevant Interrupts :

#INT_PWM1	PWM Timebase Interrupt
-----------	------------------------

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

None

Example Code:

```
// Sets up the motor PWM module
```

```

setup_motor_pwm(1,MPWM_FREE_RUN | MPWM_SYNC_OVERRIDES, timebase);

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

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

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

```

PMP/EPMP

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

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

Relevant Functions:

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

Relevant Preprocessor:

None

Relevant Interrupts :

#INT_PMP Interrupt on read or write strobe

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

None

Example Code:

setup_pmp(PAR_ENABLE | Sets up Master mode with address lines PMA0:PMA7

PAR_MASTER_MODE_1 |

PAR_STOP_IN_IDLE,0x00FF);

If (pmp_output_full ())

{

pmp_write(next_byte);

}

Program Eeprom

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

Relevant Functions:

[read_program_eeprom\(address\)](#)

Reads the program memory location (16 bit or 32 bit depend on the device).

[write_program_eeprom\(address, value\)](#)

Writes value to program memory location address.

[erase_program_eeprom\(address\)](#)

Erases FLASH_ERASE_SIZE bytes in program memory.

[write_program_memory\(address, dataptr, count\)](#)

Writes count bytes to program memory from dataptr to address. When address is a mutiple of FLASH_ERASE_SIZE an erase also performed.

[read_program_memory\(address, dataptr, count\)](#)

Read count bytes from program memory at address to dataptr.

[write_rom_memory\(address, dataptr, count\)](#)

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

[read_rom_memory \(address, dataptr, count\)](#)

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

Relevant Preprocessor:

[#ROM address={list}](#)

Can be used to put program memory data into the hex file.

[#DEVICE\(WRITE_EEPROM=ASYNC\)](#)

Can be used with #DEVICE to prevent the write function from

hanging. When this is used make sure the eeprom is not written both inside and outside the ISR.

Relevant Interrupts:

INT_EEPROM

Interrupt fires when eeprom write is complete.

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters

PROGRAM_MEMORY

Size of program memory

READ_PROGRAM

Returns 1 if program memory can be read

FLASH_WRITE_SIZE

Smallest number of bytes written in flash

FLASH_ERASE_SIZE

Smallest number of bytes erased in flash

Example Code:

```
#ROM 0x300={1,2,3,4}                                // Inserts this data into the hex file.  
erase_program_eeprom(0x00000300);                  // Erases 32 instruction locations starting at 0x0300  
write_program_eeprom(0x00000300,0x123456);           // Writes 0x123456 to 0x0300  
value=read_program_eeprom(0x00000300);               // Reads 0x0300 returns 0x123456  
write_program_memory(0x00000300,data,12);           // Erases 32 instructions starting  
                                         // at 0x0300 (multiple of erase block)  
                                         // Writes 12 bytes from data to 0x0300  
read_program_memory(0x00000300,value,12);           // reads 12 bytes to value from 0x0300
```

For chips where getenv("FLASH_ERASE_SIZE") > getenv("FLASH_WRITE_SIZE")

WRITE_PROGRAM_EEPROM

- Writes 3 bytes, does not erase (use ERASE_PROGRAM_EEPROM)

WRITE_PROGRAM_MEMORY

- Writes any number of bytes, will erase a block whenever the (lowest) byte in a block is written to. If the first address is not start of a block that block is not erased

- While writing, every fourth byte will be ignored. Fill ignored with 0x00. This is due to the 32 bit addressing and 24 bit instruction length on the PCD devices.

WRITE_ROM_MEMORY

- Writes any number of bytes, will erase a block whenever the (lowest) byte in a block is written to. If the first address is not start of a block that block is not erased.

ERASE_PROGRAM_EEPROM

- Erases a block of size FLASH_ERASE_SIZE. The lowest address bits are not used.

For chips where getenv("FLASH_ERASE_SIZE") = getenv("FLASH_WRITE_SIZE")

WRITE_PROGRAM_EEPROM

- Writes 3 bytes, no erase is needed.

WRITE_PROGRAM_MEMORY

- Writes any numbers of bytes, bytes outside the range of the write block are not changed. No erase is needed.

- While writing, every fourth byte will be ignored. Fill ignored with 0x00. This is due to the 32 bit addressing and 24 bit instruction length on the PCD devices.

WRITE_ROM_MEMORY

- Writes any numbers of bytes, bytes outside the range of the write block are not changed. No erase is needed.

ERASE_PROGRAM_EEPROM

- Erase a block of size FLASH_ERASE_SIZE. The lowest address bits are not used.

QEI

The Quadrature Encoder Interface (QEI) module provides the interface to incremental encoders for obtaining mechanical positional data.

Relevant Functions:

setup_qei(options, filter,maxcount)	Configures the QEI module.
qeい_status()	Returns the status of the QUI module.
qeい_set_count(value)	Write a 16-bit value to the position counter.
qeい_get_count()	Reads the current 16-bit value of the position counter.

Relevant Preprocessor:

None

Relevant Interrupts :

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

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

None

Example Code:

```
int16 Value;  
setup_qei(QEI_MODE_X2 | QEI_TIMER_INTERNAL,  
QEI_FILTER_DIV_2,QEI_FORWARD);  
  
Value = qei_get_count( );
```

Setup the QEI module
Read the count.

RS232 I/O

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

Relevant Functions:

getc() or getch() getchar() or fgetc()	Gets a character on the receive pin(from the specified stream in case of fgetc, stdin by default). Use KBHIT to check if the character is available.
gets() or fgets()	Gets a string on the receive pin(from the specified stream in case of fgets, STDIN by default). Use getc to receive each character until return is encountered.

<u>putc()</u> or putchar() or fputc()	Puts a character over the transmit pin(on the specified stream in the case of fputc, stdout by default)
<u>puts()</u> or fputs()	Puts a string over the transmit pin(on the specified stream in the case of fputs, stdout by default). Uses putc to send each character.
<u>printf()</u> or fprintf()	Prints the formatted string(on the specified stream in the case of fprintf, stdout by default). Refer to the printf help for details on format string.
<u>kbit()</u>	Return true when a character is received in the buffer in case of hardware RS232 or when the first bit is sent on the RCV pin in case of software RS232. Useful for polling without waiting in getc.

[setup_uart\(baud,\[stream\]\)](#)

or

[setup_uart_speed\(baud,\[stream\]\)](#) Used to change the baud rate of the hardware UART at run-time. Specifying stream is optional. Refer to the help for more advanced options.

[assert\(condition\)](#)

Checks the condition and if false prints the file name and line to STDERR. Will not generate code if #DEFINE NODEBUG is used.

[perror\(message\)](#)

Prints the message and the last system error to STDERR.

Relevant Preprocessor:

[#USE RS232\(options\)](#) This directive tells the compiler the baud rate and other options like transmit, receive and enable pins. Please refer to the #USE RS232 help for more advanced options. More than one RS232 statements can be used to specify different streams. If stream is not specified the function will use the last #USE RS232.

putc_send() or fputc_send() When using transmit buffer, used to transmit data from buffer. See function description for more detail on when needed.

rcv_buffer_bytes() When using receive buffer, returns the number of bytes in buffer that still need to be retrieved.

tx_buffer_bytes When using transmit buffer, returns the number of bytes in buffer that still need to be sent.

[tx_buffer_full\(\)](#) When using transmit buffer, returns TRUE if transmit buffer is full.

Relevant Interrupts:

INT_RDA Interrupt fires when the receive data available

INT_TBE Interrupt fires when the transmit data empty

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

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

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

Example Code:

```
/* configure and enable uart, use first hardware UART on PIC */
```

```
#use rs232(uart1, baud=9600)
```

```
/* print a string */
```

```
printf("enter a character");
```

```
/* get a character */
```

```
if (kbhit()) //check if a character has been received
```

```
c = getc(); //read character from UART
```

RTCC

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

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

Relevant Functions:

<u>setup_RTC(options, calibration);</u>	This will setup the RTCC module for operation and also allows for calibration setup.
<u>rtc_write(rtctime_t datetime)</u>	Writes the date and time to the RTCC module.
<u>rtc_read(rtctime_t datetime)</u>	Reads the current value of Time and Date from the RTCC module.
<u>setup_RTC_alarm(options, mask, repeat);</u>	Configures the alarm of the RTCC module.
<u>rtc_alarm_write(rtctime_t datetime);</u>	Writes the date and time to the alarm in the RTCC module.
<u>rtc_alarm_read(rtctime_t datetime);</u>	Reads the date and time to the alarm in the RTCC module.

Relevant Preprocessor:

None

Relevant Interrupts :

#INT_RTC Interrupt on Alarm Event or half alarm frequency.

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

None

Example Code:

setup_RTC(RTC_ENABLE | Enable RTCC module with seconds clock and no calibration.

RTC_OUTPUT_SECONDS,

0x00);

rtc_write(datetime); Write the value of Date and Time to the RTC module

rtc_read(datetime); Reads the value to a structure time_t.

RTOS

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

Relevant Functions:

[rtos_run\(\)](#) Begins the operation of the RTOS. All task management tasks are implemented by this function.

[rtos_terminate\(\)](#) This function terminates the operation of the RTOS and returns operation to the original program. Works as a return from the rtos_run()function.

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

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

[rtos_msg_poll\(\)](#) Returns true if there is data in the task's message queue.

[rtos_msg_read\(\)](#) Returns the next byte of data contained in the task's message queue.

[rtos_msg_send\(task,byte\)](#) Sends a byte of data to the specified task. The data is placed in the receiving task's message queue.

[rtos_yield\(\)](#) Called with in one of the RTOS tasks and returns control

of the program to the rtos_run() function. All tasks should call this function when finished.

[rtos_signal\(sem\)](#) Increments a semaphore which is used to broadcast the availability of a limited resource.

[rtos_wait\(sem\)](#) Waits for the resource associated with the semaphore to become available and then decrements the semaphore to claim the resource.

[rtos_await\(expr\)](#) Will wait for the given expression to evaluate to true before allowing the task to continue.

[rtos_overrun\(task\)](#) Will return true if the given task overran its allotted time.

[rtos_stats\(task,stat\)](#) Returns the specified statistic about the specified task. The statistics include the minimum and maximum times for the task to run and the total time the task has spent running.

Relevant Preprocessor:

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

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

[#TASK](#) specifies the rate at which the task should be called, the maximum time the task shall be allowed to run, and how large its queue should be

Relevant Interrupts:

none

Relevant Include Files:

none all functions are built in

Relevant getenv() Parameters:

none

Example Code:

```
#USE RTOS(timer=0,minor_cycle=20ms) // RTOS will use timer zero, minor cycle will be 20ms
```

...

```
int sem;
```

...

```
#TASK(rate=1s,max=20ms,queue=5) // Task will run at a rate of once per second  
void task_name(); // with a maximum running time of 20ms and
```

```
// a 5 byte queue
```

```
rtos_run(); // begins the RTOS  
rtos_terminate(); // ends the RTOS
```

```
rtos_enable(task_name); // enables the previously declared task.  
rtos_disable(task_name); // disables the previously declared task
```

```

rtos_msg_send(task_name,5);           // places the value 5 in task_names queue.
rtos_yield();                      // yields control to the RTOS
rtos_sigal(sem);                  // signals that the resource represented by sem is
                                    available.

```

For more information on the CCS RTOS please

SPI

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

Relevant Functions:

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

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

spi_data_is_in()	Returns TRUE if the SPI receive buffer has a byte of data.
spi_data_is_in2()	

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

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

Relevant Preprocessor:

None

Relevant Interrupts:

#int_ssp	Transaction (read or write) has completed on the indicated peripheral.
#int_ssp2	

#int_spi1	Interrupts on activity from the first SPI module
#int_spi2	Interrupts on activity from the second SPI module

Relevant Include Files:

None, all functions built-in to the compiler.

Relevant getenv() Parameters:

SPI	Returns TRUE if the device has an SPI peripheral
-----	--

Example Code:

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

```

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

```

Timers

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

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

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

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

Relevant Functions:

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

Relevant Preprocessor:

None

Relevant Interrupts:

<code>#int_timerX</code>	Interrupts on timer overflow (period match). X is any valid timer number.
--------------------------	---

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

Relevant Include Files:

None, all functions built-in

Relevant `getenv()` parameters:

<code>TIMERX</code>	Returns 1 if the device has the timer peripheral X. X may be 1 - 9
---------------------	--

Example Code:

```
/* Setup timer1 as an external real time clock that increments every 16 clock cycles */
```

```

setup_timer1(T1_EXTERNAL_RTC | T2_DIV_BY_16 );
/* Setup timer2 as a timer that increments on every instruction cycle and has a period of 0x0100 */
setup_timer2(TMR_INTERNAL, 0x0100);
byte value = 0x00;
value = get_timer2();           //retrieve the current value of timer2

```

TimerA

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

Relevant Functions:

setup_timer_A(mode)	Disable or sets the source and prescale for timerA
set_timerA(value)	Initializes the timerA clock/counter
value=get_timerA()	Returns the value of the timerA clock/counter

Relevant Preprocessor:

None

Relevant Interrupts :

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

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

TIMERA	Returns 1 if the device has timerA
--------	------------------------------------

Example Code:

```

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

```

TimerB

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

Relevant Functions:

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

Relevant Preprocessor:

None

Relevant Interrupts :

INT_TIMERB	Interrupt fires when timerB overflows
------------	---------------------------------------

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

TIMERB Returns 1 if the device has timerB

Example Code:

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

Voltage Reference

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

Relevant Functions:

[setup_vref\(mode| value\)](#) Enables and sets up the internal voltage reference value.
Constants are defined in the devices .h file.

Relevant Preprocessor:

None

Relevant Interrupts:

None

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

VREF Returns 1 if the device has VREF

Example Code:

For eg:

For PIC12F675

#INT_COMP //comparator interrupt handler

void isr() {

safe_conditions=FALSE;

printf("WARNING!! Voltage level is above 3.6 V. \r\n");

}

setup_comparator(A1_VR_OUT_ON_A2); // sets two comparators(A1 and VR and A2 as the output)

setup_vref(VREF_HIGH|15); //sets 3.6(vdd *value/32 +vdd/4) if vdd is 5.0V

enable_interrupts(INT_COMP); //enables the comparator interrupt

enable_interrupts(GLOBAL); //enables global interrupts

WDT or Watch Dog Timer

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

Relevant Functions:

<u>setup_wdt()</u>	Enables/disables the wdt or sets the prescalar.
<u>restart_wdt()</u>	Restarts the wdt, if wdt is enabled this must be periodically called to prevent a timeout reset.

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

The timeout time for PCB/PCM chips are set using the setup_wdt function and on PCH using fuses like WDT16, WDT256 etc.

RESTART_WDT when specified in #USE DELAY, #USE I2C and #USE RS232 statements like this #USE DELAY(clock=20000000, restart_wdt) will cause the wdt to restart if it times out during the delay or I2C_READ or GETC.

Relevant Preprocessor:

<u>#FUSES WDT/NOWDT</u>	Enabled/Disables wdt in PCB/PCM devices
<u>#FUSES WDT16</u>	Sets up the timeout time in PCH devices

Relevant Interrupts:

None

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

None

Example Code:

For eg:

For PIC16F877

```
#fuses wdt
setup_wdt(WDT_2304MS);
while(true){
    restart_wdt();
    perform_activity();
}
```

For PIC18F452

```
#fuse WDT1
setup_wdt(WDT_ON);
while(true){
    restart_wdt();
    perform_activity();
}
```

Some of the PCB chips share the WDT prescalar bits with timer0 so the WDT prescalar constants can be used with setup_counters or setup_timer0 or setup_wdt functions.

PRE-PROCESSOR

#ASM #ENDASM

Syntax:	#ASM or #ASM ASIS code #ENDASM
Elements:	code is a list of assembly language instructions
Examples:	<pre>int_ffind_parity(int data){int count;, result,data1; data1=data; asm MOV #0x08, MOV WF, count CLRF result Loop: MOVF data1,w XORWF result, F RRCF data1,F DECSZ count,F BRA LOOP MOVLW 0x01 ANDWF resultt, F #end asm return (result); }</pre>
Example Files:	ex_glint.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	Wd = f+Wa+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 â€“ Wa
CP	Wd,lit5	Status set for Wa â€“ lit5
CP.B	W,f	Status set for f - W0 (byte)
CP.B	Wa,Wd	Status set for Wb â€“ Wa (byte)
CP.B	Wd,lit5	Status set for Wa â€“ lit5 (byte)
CP0	Wd	Status set for Wa â€“ 0
CP0	W,f	Status set for f â€“ 0
CP0.B	Wd	Status set for Wa â€“ 0 (byte)
CP0.B	W,f	Status set for f â€“ 0 (byte)
CPB	Wd,lit5	Status set for Wa â€“ lit5 â€“ C
CPB	Wa,Wd	Status set for Wb â€“ Wa â€“ C
CPB	W,f	Status set for f â€“ W0 - C
CPB.B	Wa,Wd	Status set for Wb â€“ Wa â€“ C (byte)
CPB.B	Wd,lit5	Status set for Wa â€“ lit5 â€“ C (byte)
CPB.B	W,f	Status set for f â€“ 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	$f = f \mid Wa$ (byte)
LAC	Wd,{lit4},acc	Acc = Wa shifted slit4
LNK	lit14	Allocate Stack Frame
LSR	f,W	$W0 = f \gg 1$
LSR	Wa,lit4,Wd	$Wd = Wa \gg \text{lit4}$
LSR	Wa,Wd	$Wd = Wa \gg 1$
LSR	f,F	$f = f \gg 1$
LSR	Wa,Wb,Wd	$Wd = Wb \gg Wa$
LSR.B	f,W	$W0 = f \gg 1$ (byte)
LSR.B	f,F	$f = f \gg 1$ (byte)
LSR.B	Wa,Wd	$Wd = Wa \gg 1$ (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	$Wd = [Wa + \text{Slit10}]$
MOV	? ,Wd	$Wd = f$
MOV	lit16,Wd	$Wd = \text{lit16}$
MOV	Wa,Wd	$Wd = Wa$
MOV	Wa,Wd+lit	$[Wd + \text{Slit10}] = Wa$
MOV.B	lit8,Wd	$Wd = \text{lit8}$ (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	$Wd = [Wa + \text{Slit10}]$ (byte)
MOV.B	Wa,Wd+lit	$[Wd + \text{Slit10}] = Wa$ (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 - 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
PWRSAV	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)

SSTEP		ICD Single Step
SUB	f,F	$f = f \leftarrow W_0$
SUB	f,W	$W_0 = f \leftarrow W_0$
SUB	Wa,Wb,Wd	$W_d = Wa \leftarrow W_b$
SUB	Wa,lit5,Wd	$W_d = Wa \leftarrow \text{lit5}$
SUB	acc	$\text{Acc} = \text{AccA} \leftarrow \text{AccB}$
SUB	lit10,Wd	$W_d = W_d \leftarrow \text{lit10}$
SUB.B	Wa,lit5,Wd	$W_d = Wa \leftarrow \text{lit5 (byte)}$
SUB.B	lit10,Wd	$W_d = W_d \leftarrow \text{lit10 (byte)}$
SUB.B	f,W	$W_0 = f \leftarrow W_0 \text{ (byte)}$
SUB.B	Wa,Wb,Wd	$W_d = Wa \leftarrow W_b \text{ (byte)}$
SUB.B	f,F	$f = f \leftarrow W_0 \text{ (byte)}$
SUBB	f,W	$W_0 = f \leftarrow W_0 \leftarrow C$
SUBB	Wa,Wb,Wd	$W_d = Wa \leftarrow W_b \leftarrow C$
SUBB	f,F	$f = f \leftarrow W_0 \leftarrow C$
SUBB	Wa,lit5,Wd	$W_d = Wa \leftarrow \text{lit5} - C$
SUBB	lit10,Wd	$W_d = W_d \leftarrow \text{lit10} \leftarrow C$
SUBB.B	lit10,Wd	$W_d = W_d \leftarrow \text{lit10} \leftarrow C \text{ (byte)}$
SUBB.B	Wa,Wb,Wd	$W_d = Wa \leftarrow W_b \leftarrow C \text{ (byte)}$
SUBB.B	f,F	$f = f \leftarrow W_0 \leftarrow C \text{ (byte)}$
SUBB.B	Wa,lit5,Wd	$W_d = Wa \leftarrow \text{lit5} - C \text{ (byte)}$
SUBB.B	f,W	$W_0 = f \leftarrow W_0 \leftarrow C \text{ (byte)}$
SUBBR	Wa,lit5,Wd	$W_d = \text{lit5} \leftarrow Wa - C$
SUBBR	f,W	$W_0 = W_0 \leftarrow f \leftarrow C$
SUBBR	f,F	$f = W_0 \leftarrow f \leftarrow C$
SUBBR	Wa,Wb,Wd	$W_d = Wa \leftarrow W_b - C$
SUBBR.B	f,F	$f = W_0 \leftarrow f \leftarrow C \text{ (byte)}$
SUBBR.B	f,W	$W_0 = W_0 \leftarrow f \leftarrow C \text{ (byte)}$
SUBBR.B	Wa,Wb,Wd	$W_d = Wa \leftarrow W_b - C \text{ (byte)}$
SUBBR.B	Wa,lit5,Wd	$W_d = \text{lit5} \leftarrow Wa - C \text{ (byte)}$
SUBR	Wa,lit5,Wd	$W_d = \text{lit5} \leftarrow W_b$
SUBR	f,F	$f = W_0 \leftarrow f$
SUBR	Wa,Wb,Wd	$W_d = Wa \leftarrow W_b$
SUBR	f,W	$W_0 = W_0 \leftarrow f$
SUBR.B	Wa,Wb,Wd	$W_d = Wa \leftarrow W_b \text{ (byte)}$
SUBR.B	f,F	$f = W_0 \leftarrow f \text{ (byte)}$
SUBR.B	Wa,lit5,Wd	$W_d = \text{lit5} \leftarrow W_b \text{ (byte)}$
SUBR.B	f,W	$W_0 = W_0 \leftarrow f \text{ (byte)}$
SWAP	Wd	$Wa = \text{byte or nibble swap } Wa$
SWAP.B	Wd	$Wa = \text{byte or nibble swap } Wa \text{ (byte)}$
TBLRDH	Wa,Wd	$W_d = \text{ROM}[Wa] \text{ for odd ROM}$
TBLRDH.B	Wa,Wd	$W_d = \text{ROM}[Wa] \text{ for odd ROM (byte)}$
TBLRDL	Wa,Wd	$W_d = \text{ROM}[Wa] \text{ for even ROM}$
TBLRDL.B	Wa,Wd	$W_d = \text{ROM}[Wa] \text{ for even ROM (byte)}$
TBLWTH	Wa,Wd	$\text{ROM}[Wa] = W_d \text{ for odd ROM}$
TBLWTH.B	Wa,Wd	$\text{ROM}[Wa] = W_d \text{ for odd ROM (byte)}$
TBLWTL	Wa,Wd	$\text{ROM}[Wa] = W_d \text{ for even ROM}$
TBLWTL.B	Wa,Wd	$\text{ROM}[Wa] = W_d \text{ for even ROM (byte)}$
ULNK		Deallocate Stack Frame
URUN		ICD Run

XOR	Wa,Wb,Wd	Wd = Wa ^ Wb
XOR	f,F	f = f ^ W0
XOR	f,W	W0 = f ^ W0
XOR	Wa,lit5,Wd	Wd = Wa ^ lit5
XOR	lit10,Wd	Wd = Wd ^ lit10
XOR.B	lit10,Wd	Wd = Wd ^ lit10 (byte)
XOR.B	f,W	W0 = f ^ W0 (byte)
XOR.B	Wa,lit5,Wd	Wd = Wa ^ lit5 (byte)
XOR.B	Wa,Wb,Wd	Wd = Wa ^ Wb (byte)
XOR.B	f,F	f = f ^ W0 (byte)
ZE	Wa,Wd	Wd = Wa & FF

12 Bit and 14 Bit

ADDWF f,d	ANDWF f,d
CLRF f	CLRW
COMF f,d	DECf f,d
DECFSZ f,d	INCF f,d
INCFSZ f,d	IORWF f,d
MOVF f,d	MOVPHW
MOVPLW	MOVWF f
NOP	RLF f,d
RRF f,d	SUBWF f,d
SWAPF f,d	XORWF f,d
BCF f,b	BSF f,b
BTFSZ f,b	BTFSZ f,b
ANDLW k	CALL k
CLRWD	GOTO k
IORLW k	MOVLW k
RETLW k	SLEEP
XORLW	OPTION
TRIS k	
	14 Bit
	ADDLW k
	SUBLW k
	RETFIE
	RETURN

PIC 18

ADDWF	f,d	ADDWFCf,d	ANDWF f,d
CLRF	f	COMF f,d	CPFSEQ f
CPFSGT	f	CPFSLT f	DECf f,d
DECFSZ	f,d	DCFSNZ f,d	INCF f,d
INFSNZ	f,d	IORWF f,d	MOVF f,d
MOVFF	fs,d	MOVWF f	MULWF f
NEGF	f	RLCF f,d	RLNCF f,d
RRCF	f,d	RRNCF f,d	SETF f
SUBFWB	f,d	SUBWF f,d	SUBWFBf,d
SWAPF	f,d	TSTFSZ f	XORWF f,d
BCF	f,b	BSF f,b	BTFSZ f,b

BTFSS	f,b	BTG	f,dBC	n
BN	n	BNC	n BNN	n
BNOV	n	BNZ	n BOV	n
BRA	n	BZ	n CALL	n,s
CLRWDT	-	DAW	- GOTO	n
NOP	-	NOP	- POP	-
PUSH	-	RCALL	n RESET	-
RETFIE	s	RETLW	k RETURN	s
SLEEP	-	ADDLW	k ANDLW	k
IORLW	k	LFSR	f,k MOVLB	k
MOVLW	k	MULLW	k RETLW	k
SUBLW	k	XORLW	k TBLRD	*
TBLRD	*+	TBLRD	*- TBLRD	+-
TBLWT	*	TBLWT	*+ TBLWT	*-
TBLWT	+-			

#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:	<pre>#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</pre>
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 None

Files:

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 None

Files:

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 T 1 IF = 0x 84.3
...
TSB:1IF = 0; // Clear Timer 0 interrupt flag

int result;
#bit result_odd = result.0

```
...  
if (result_odd)
```

Example [ex_glint.c](#)

Files:

Also See: [#BYTE](#), [#RESERVE](#), [#LOCATE](#), [#WORD](#)

#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)

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

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

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

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

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

no sleep is used to prevent the compiler from inserting a sleep at the end of main()

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 */
```

```

#define(reset=0x200:0x207, interrupt=0x208:0x2ff)

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

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

```

Example 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](#)

#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 compile in the form: "31-JAN-03"

Examples:

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

Example None

Files:

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*** and so on are 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 idx 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

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"
```

Example [ex_stwt.c](#), [ex_macro.c](#)
Files:

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 – typmod 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

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.

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

Compilation mode selection-

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

CCS4	This is the default compilation mode.
ANSI	Default data type is SIGNED all other modes default is UNSIGNED. Compilation is case sensitive, all other modes are case insensitive.
CCS2	var16 = NegConst8 is compiled as: var16 = NegConst8 & 0xff (no sign extension) . The overload keyword is required.
CCS3	
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](#)

#ERROR

Syntax: **#ERROR *text***
#ERROR / warning *text*
#ERROR / information *text*

Elements: ***text*** is optional and may be any text

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

Examples:

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

Example [ex_psp.c](#)

Files:

Also See: [#WARNING](#)

#EXPORT (options)

Syntax: #EXPORT (options)

Elements:	FILE=filename The filename which will be generated upon compile. If not given, the filename will be the name of the file you are compiling, with a .o or .hex extension (depending on output format).
	ONLY=symbol+symbol+.....+symbol Only the listed symbols will be visible to modules that import or link this relocatable object file. If neither ONLY or EXCEPT is used, all symbols are exported.
	EXCEPT=symbol+symbol+.....+symbol All symbols except the listed symbols will be visible to modules that import or link this relocatable object file. If neither ONLY or EXCEPT is used, all symbols are exported.
	RELOCATABLE CCS relocatable object file format. Must be imported or linked before loading into a PIC. This is the default format when the #EXPORT is used.
	HEX Intel HEX file format. Ready to be loaded into a PIC. This is the default format when no #EXPORT is used.
	RANGE=start:stop Only addresses in this range are included in the hex file.
	OFFSET=address Hex file address starts at this address (0 by default)
	ODD Only odd bytes place in hex file.
	EVEN Only even bytes placed in hex file.

Purpose: This directive will tell the compiler to either generate a relocatable object file or a stand-alone HEX binary. A relocatable object file must be linked into your application, while a stand-alone HEX binary can be programmed directly into the PIC. The command line compiler and the PCW IDE Project Manager can also be used to compile/link/build modules and/or projects.

Multiple #EXPORT directives may be used to generate multiple hex files. this may be used for 8722 like devices with external memory.

Examples:

```
#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 Files: None

See Also: [#IMPORT](#), [#MODULE](#), [Invoking the Command Line Compiler](#), [Multiple Compilation Unit](#)

__FILENAME__

Syntax: __FILENAME__

Elements: None

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

Examples:

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

Example Files: None

Also See: [line](#)

#FILL_ROM

Syntax: **#fill_rom value**

Elements: **value** is a constant 16-bit value

Purpose: This directive specifies the data to be used to fill unused ROM locations. When linking multiple

compilation units, this directive must appear exactly the same in each compilation unit.

Examples: `#fill_rom 0x36`

Example Files: None

Also See: [#ROM](#)

#FUSES

Syntax: `#FUSES options`

Elements: ***options*** vary depending on the device. A list of all valid options has been put at the top of each 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 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

#IF exp #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\(\)](#)

#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

...
#ifndef DEBUG
printf("debug point a");
#endif
```

Example Files: [ex_sqw.c](#)

Also See: [#IF](#)

#IGNORE_WARNINGS

Syntax:

```
#ignore_warnings ALL
#ifndef IGNORE_WARNINGS
#define IGNORE_WARNINGS 0
#endif
```

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

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_ad
adc_handler() {
    adc_active=FALSE;
}

#int_timer1 noclear
isr() {
    ...
}
```

Example None

Files:

Also See: [enable_interrupts\(\)](#), [disable_interrupts\(\)](#),
[#INT_DEFAULT](#),

#INT_DEFAULT

Syntax: #INT_DEFAULT

Elements: None

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

Examples:

```
#int_default
default_isr() {
    printf("Unexplained
interrupt\r\n");
}
```

Example None

Files:

Also See: #INT_xxxx,

__LINE__

Syntax: __line__

Elements: None

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

Examples:

```
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 [None](#)

Files:

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 [ex_glint.c](#)

Files:

Also See: [#BYTE](#), [#BIT](#), [#RESERVE](#), [#WORD](#)

#MODULE

Syntax: #MODULE

Elements: None

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

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

Examples:

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

Example None

Files:

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

#NOLIST

Syntax: #NOLIST

Elements: None

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

Examples:

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

Example [16c74.h](#)

Files:

Also See: [#LIST](#)

#OCS

Syntax: #OCS x

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

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

Examples:

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

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

Example None

Files:

Also See: [#USE DELAY](#)

#OPT

Syntax: #OPT n

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: #opt 5

Example None

Files:

Also See: None

#ORG

Syntax: `#ORG start, end`

 or

`#ORG segment`

 or

`#ORG start, end { }`

 or

`#ORG start, end auto=0`

`#ORG start,end DEFAULT`

 or

`#ORG DEFAULT`

Elements: **start** is the first ROM location (word address) to use,
end is the last ROM location, **segment** is the start
ROM location from a previous #ORG

Purpose: This directive will fix the following function, constant or
ROM declaration into a specific ROM area. End may
be omitted if a segment was previously defined if you
only want to add another function to the segment.

Follow the ORG with a { } to only reserve the area with
nothing inserted by the compiler.

The RAM for a ORG'd function may be reset to low
memory so the local variables and scratch variables
are placed in low memory. This should only be used if
the ORG'd function will not return to the caller. The
RAM used will overlap the RAM of the main
program. Add a AUTO=0 at the end of the #ORG line.

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

ROM. #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
CSCKOUT	DCI Serial Clock Output
COFSOUT	DCI Frame Sync Output
UPDN1	QEI1 Direction Status Output
UPDN2	QEI2 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
QEI1CCMP	QEI1 Counter Comparator Output
QEI2CCMP	QEI2 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
CSCK	DCI Serial Clock Input
COFS	DCI Frame Sync Input
FLTA1	PWM1 Fault Input
FLTA2	PWM2 Fault Input
QEA1	QEI1 Phase A Input
QEA2	QEI2 Phase A Input
QEB1	QEI1 Phase B Input
QEB2	QEI2 Phase B Input
INDX1	QEI1 Index Input
INDX2	QEI2 Index Input
HOME1	QEI1 Home Input
HOME2	QEI2 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
SYNC1	PWM Synchronization Input 1

SYNCI2	PWM Synchronization Input 2
DCIDI	DCI Serial Data Input
DCISCKIN	DCI Serial Clock Input
DCIF SIN	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 **cmd**

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

Purpose: This directive is used to maintain compatibility between C compilers. This compiler will accept this directive before any other pre-processor command. In no case does this compiler require this directive.

Examples: #pragma device PIC16C54

Example [ex_cust.c](#)

Files:

Also See: None

#PROFILE

Syntax: #profile options

Element **options** may be one of the following:

s:

functions	Profiles the start/end of functions and all profileout() messages.
functions, parameters	Profiles the start/end of functions, parameters sent to functions, profileout() messages.
profileout	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 off command. This will use the last options before disabled with 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.

Example #profile off
s:
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 [ex_profile.c](#)

Files:

Also [#use profile\(\)](#), [profileout\(\)](#), [Code Profile overview](#)

See:

#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 [None](#)

Files:

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

Example None

Files:

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:	<p><i>id=xxx</i> - Specify a C CONST identifier, may be int8, int16, int32 or char array</p>
	<p>Use in place of id parameter, when storing serial number to EEPROM: <i>dataee=x</i> - The address x is the start address in the data EEPROM. <i>binary=x</i> - The integer x is the number of bytes to be written to address specified. -or- <i>string=x</i> - The integer x is the number of bytes to be written to address specified.</p>
	<p>Use only one of the next three options: <i>file="filename.txt"</i> - 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.</p> <p><i>listfile="filename.txt"</i> - 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.</p> <p><i>next="x"</i> - The serial number X is used for the first load, then the hex file is updated to increment x by one.</p>
	<p>Other optional parameters: <i>prompt="text"</i> - 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.</p> <p><i>log=xxx</i> - 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.</p>
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:	<pre>//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</pre>

```

    #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

Stream I/O

Syntax: #include <iostream.h> is required to use any of the ios identifiers.

Output: output:
`stream << variable_or_constant_or_manipulator ;`

one or more repeats

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

stream may also be the name of a char array. In this case the data is
written to the array with a 0 terminator.

stream may also be the name of a function that accepts a single char
parameter. In this case the function is called for each character to be output.

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

manipulators:

hex -Hex format numbers

dec- Decimal format numbers (default)

setprecision(x) -Set number of places after the decimal point

setw(x) -Set total number of characters output for numbers

boolalpha- Output int1 as true and false

noboolalpha -Output int1 as 1 and 0 (default)

fixed Floats- in decimal format (default)

scientific Floats- use E notation

iosdefault- All manipulators to default settings

endl -Output CR/LF

ends- Outputs a null ('\000')

Examples: cout << "Value is " << hex << data << endl;

cout << "Price is \$" << setw(4) << setprecision(2) << cost << endl;

Lcdputc << 'f' << setw(3) << count << " " << min << " " << max;

```
string1 << setprecision(1) << sum / count;  
string2 << x << ',' << y;
```

Input: stream >> variable_or_constant_or_manipulator ;

one or more repeats

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

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

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

variables/constants: May be any integer, char, float or fixed type. Char arrays are
input as strings. Floats may use the E format.

Reading of each item terminates with any character not valid for the type. Usually
items are separated by spaces. The termination character is discarded. At the end
of any stream input statement characters are read until a return (\r) is read. No
termination character is read for a single char input.

manipulators:

hex -Hex format numbers

dec- Decimal format numbers (default)

noecho- Suppress echoing

strspace- Allow spaces to be input into strings

nostrspace- Spaces terminate string entry (default)

iosdefault -All manipulators to default settings

Examples: cout << "Enter number: ";
cin >> value;
cout << "Enter title: ";
cin >> strspace >> title;
cin >> data[i].recordid >> data[i].xpos >> data[i].ypos >> data[i].sample ;
string1 >> data;
lcdputc << "\fEnter count";
lcdputc << keypadgetc >> count; // read from keypad, echo to lcd
// This syntax only works with
// user defined functions.

#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 None

Files:

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](#)

#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 if 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.

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 [ex_malloc.c](#)

Files:

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 [ex_cust.c](#)

Files:

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 devices .h file.

Purpose: This directive affects how the compiler will generate code for input and output

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

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

Example None

Files:

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 i2c_read or i2c_write.
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 [ex_extee.c](#) with [16c74.h](#)

Files:

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)`

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

S:

ICD	Required – configures code profiler to use the ICD connection. Optional. If specified, the code profiler run-time on the microcontroller will use the Timer1 peripheral as a timestamp for all profile events. If no timer is specified the code profiler tool will use the PC clock, which may not be accurate for fast events.
TIMER 1	Optional. If specified, will use a different baud rate between the microcontroller and the code profiler tool. This may be required for slow microcontrollers to attempt to use a slower baud rate.
BAUD=x	Optional. If specified, will use a different baud rate between the microcontroller and the code profiler tool. This may be required for 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.

Example `#profile(ICD, TIMER1, baud=9600)`

S:

Example [ex_profile.c](#)

Files:

Also [#profile\(\)](#), [profileout\(\)](#), [Code Profile overview](#)

See:

#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 OC pins. If device has remappable pins compiler will assign specified pin to specified OC module. If OC module not specified it will assign remappable pin to first available module.
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%.
STREAM=id	Associates a stream identifier with the PWM signal. The identifier may be used in functions like <code>pwm_set_duty_percent()</code> .

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().

Example Files 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 in 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 from 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 setup_uart() needs to be used to initialize the peripheral. Using a serial routine (such as getc() or putc()) before the UART is initialized will cause undefined behavior.
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 than 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 than 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 than 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 used 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 is 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.

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

Purpose: 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 be 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 prescalar to setup counter with, default is 1 if x is not specified specified. The function get_ticks() will return the current count and the function set_ticks() can be used to set count to a specific starting value or to clear counter.

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 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.

Examples: #if BUFFER_SIZE < 32
#warning Buffer Overflow may occur
#endif

Example [ex_psp.c](#)

Files:

Also See: [#ERROR](#)

#WORD

Syntax: #WORD *id* = *x*

Elements: *id* is a valid C identifier,

x is a C variable or a constant

Purpose: If the id is already known as a C variable then this will locate the variable at address x. In this case the variable type does not change from the original definition. If the id is not known a new C variable is created and placed at address x with the type int16

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

Examples: #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](#)

#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 [ex_cust.c](#)

Files:

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.

RS232 I/O

SPI

TWO WIRE I/O

DISCRETE I/O	get_tris_x() input() input_state() set_tris_x() input_x()	output_X() output_bit() input_change_x() output_float() output_high()	output_drive() output_low() output_toggle() port_x_pullups()
PARALLEL PORT	psp_input_full() psp_overflow()	psp_output_full() setup_psp(option, address, mask)	
I2C I/O	i2c_isr_state() i2c_poll() i2c_read()	i2c_slaveaddr() i2c_start() i2c_stop()	i2c_write() i2c_speed()
PROCESSOR CONTROLS	clear_interrupt() disable_interrupts() enable_interrupts() ext_int_edge() getenv() brownout_enable()	goto_address() interrupt_active() jump_to_isr() label_address() read_bank()	reset_cpu() restart_cause() setup_oscillator() sleep() write_bank()
BIT/BYTE MANIPULATION	bit_clear() bit_set() shift_left() bit_test()	make8() make16() shift_right() make32()	mul() rotate_left() swap() rotate_right()
STANDARD C MATH	abs() acos() asin() atan() atan2() atoe() ceil() cos() cosh()	div() exp() fabs() floor() fmod() frexp() labs() ldexp() ldiv()	log() log10() modf() pow() sin() sinh() sqrt() tan() tanh()
VOLTAGE REF	setup_low_volt_detect()	setup_vref()	-
A/D CONVERSION	set_adc_channel() adc_done()	setup_adc() setup_adc_ports()	read_adc()

	atof()	isxdigit(char)	strncpy()
	atoi()	itoa()	strpbrk()
	atol32()	sprintf()	strcpy()
	atol()	strcat()	 strrchr()
	isalnum()	strchr()	strspn()
	isalpha(char)	strcmp()	strstr()
STANDARD C CHAR/STRING	isamong()	strcoll()	strtod()
	iscntrl(x)	strcpy()	strtok()
	isdigit(char)	strcspn()	 strtol()
	isgraph(x)	strerror()	strtoul()
	islower(char)	strcmp()	strxfrm()
	isprint(x)	strlen()	tolower()
	ispunct(x)	strlwr()	toupper()
	isspace(char)	strncat()	strcpy()
	isupper(char)	strncmp()	
TIMERS	get_timer_x()	set_timerx()	setup_timer_0()
	setup_timer_1()	setup_timer_2()	setup_timer_3()
	setup_timer_4()	setup_timer_5()	setup_counters()
	restart_wdt()	setup_wdt()	set_rtcc()
	set_ticks()	get_ticks()	setup_timer_A()
	setup_timer_B()	set_timerA()	set_timerB()
	get_timerA()	get_timerB()	
STANDARD C MEMORY	calloc()	memcmp()	setup_timer_0()
	free()	memcpy()	offsetofbit()
	longjmp()	memmove()	realloc()
	malloc()	memset()	setjmp()
	memchr()	offsetof()	
CAPTURE COMARE/PWM	get_capture_event()	setup_cwg()	
	set_power_pwm_override()	cwg_reset()	

	set_power_pwmx_duty()	
	seup_pwmx()	
NON-VOLATILE MEMORY	erase_eeprom()	read_external_memory()
	erase_program_eeprom()	read_program_eeprom()
	read_calibration()	read_program_memory()
	read_configuration_memory()	setup_external_memory()
	read_eeprom()	write_configuration_memory()
STANDARD C SPECIAL	bsearch()	rand()
	nargs()	srand()
	qsort()	va_arg()
DELAYS	delay_cycles()	delay_ms()
		delay_us()
ANALOG COMPARE	setup_comparator()	-
RTOS	rtos_await()	rtos_msg_send()
	rtos_disable()	rtos_overrun()
	rtos_enable()	rtos_run()
	rtos_msg_poll()	rtos_signal()
	rtos_msg_read()	rtos_stats()
LCD	lcd_contrast()	lcd_load()
	setup_lcd()	lcd_symbol()
QEI	gei_get_count()	gei_set_count()
	setup_qei()	gei_status()
CONFIGURABLE LOGIC CELL		
NUMERICALLY CONTROLLED OSCILLATOR		
D/A CONVERSION	dac_write()	setup_dac()

REAL TIME CLOCK CALENDAR	rtc_read() rtc_alarm_write()	setup_dac() setup_RTC()	rtc_alarm_read() setup_RTC_alarm()
CAPACITIVE TOUCH PAD	touchpad_getc()	touchpad_hit()	touchpad_state()
PARALLEL MASTER PORT	setup_pmp()		
MISC.	setup_opamp1() assert() fgetc() fgets() fprintf() fputc() fputs()	setup_opamp2() getch() getchar() gets() kbhit() perror() printf()	sleep_ulpwu() putc() putchar() puts() setup_uart() set_uart_speed() getc()
RS232 I/O	 setup_spi()	spi_data_is_in()	spi_read()
SPI TWO WIRE	setup_spi2() spi_xfer()	spi_data_is_in2() spi_write()	spi_read2() spi_write2()
DISCRETE I/O	 get_tris_x() input() input_change_x() input_state() input_x()	output_bit() output_drive() output_float() output_high() output_low()	output_toggle() output_x() set_pullup() set_tris_x()
I2C I/O	 i2c_isr_state()	i2c_stop()	i2c_speed()
	i2c_poll()	i2c_slaveaddr()	i2c_write()
	i2c_read()	i2c_start()	
PROCESSOR CONTROLS	clear_interrupt() disable_interrupts()	getenv() setup_oscillator()	label_address() reset_cpu()

	enable_interrupts()	goto_address()	restart_cause()
	ext_int_edge()	interrupt_active()	sleep()
	bit_clear()	bit_last()	_mul()
	bit_set()	make8()	rotate_left()
BIT/BYTE MANIPULATION	bit_test()	make16()	rotate_right()
	bit_first()	make32()	shift_left()
	shift_right()	swap()	
	abs()	cos()	log()
	acos()	cosh()	log10()
	asin()	div()	modf()
	atan()	exp()	pow()
STANDARD C MATH	atan2()	fabs()	pwr()
	atoe()	floor()	sin()
	atof48()	fmod()	sinh()
	atof64()	frexp()	sqrt()
	atoi32()	labs()	tan()
	atoi48()	ldexp()	tanh()
	ceil()	ldiv()	
VOLTAGE REF/COMP	<u>setup_comparator()</u>	<u>setup_low_volt_detect()</u>	<u>setup_vref()</u>
	adc_done()	read_adc()	set_adc_channel2()
A/D CONVERSION	adc_done2()	setup_adc2()	setup_adc_ports()
	setup_adc()	set_adc_channel()	<u>setup_adc_ports2()</u>
	atof()	isupper(char)	strncpy()
STANDARD C CHAR STRING	atoi()	isxdigit(char)	strcmp()
	atol()	itoa()	strtod()
	isalnum()	sprintf()	strchr()
	isalpha(char)	strcat()	strspn()

	isamong()	strpbrk()	strstr()
	iscntrl(x)	strcpy()	strtod()
	isdigit(char)	strcmp()	strtok()
	isgraph(x)	strcoll()	strtol()
	strerror()	strcpy()	strtoul()
	strchr()	strcspn()	strxfrm()
	islower(char)	strlen()	tolower()
	isprint(x)	strlwr()	toupper()
	ispunct(x)	strncat()	strtof48()
	isspace(char)	strncmp()	
	get_timerx()	set_timerx()	setup_wdt()
TIMERS	get_timerxy()	set_timerxy()	get_ticks()
	<u>restart_wdt()</u>	<u>setup_timerx()</u>	<u>set_ticks()</u>
	calloc()	memcmp()	offsetofbit()
	free()	memcpy()	realloc()
STANDARD C MEMORY	longjmp()	memmove()	setjmp()
	malloc()	memset()	
	memchr()	offsetof()	
	set_pwm_duty()	setup_power_pwm()	get_motor_pwm_count()
	set_motor_unit()	setup_power_pwm_pins()	set_motor_pwm_duty()
CAPTURE/COMPARE PWM	setup_motor_pwm()	set_compare_time()	get_capture()
	setup_capture()	setup_compare()	set_motor_pwm_event()
NON- VOLATILE MEMORY	erase_program_memory()	write_configuration_memory()	write_eeprom()

	read_eeprom()	read_program_memory()	write_program_memory()
	read_configuration_memory()	read_rom_memory()	
	bsearch()	rand()	va_end()
STANDARD C SPECIAL	nargs()	srand()	va_start()
	<u>qsort()</u>	<u>va_arg()</u>	
DELAYS	<u>delay_cycles()</u>	<u>delay_ms()</u>	<u>delay_us()</u>
	rtos_await()	rtos_msg_send()	rtos_terminate()
	rtos_disable()	rtos_overrun()	rtos_wait()
RTOS	rtos_enable()	rtos_run()	rtos_yield()
	rtos_msg_poll()	rtos_signal()	
	rtos_msg_read()	rtos_stats()	
DSP	TBD		
DMA	<u>dma_status()</u>	<u>dma_start()</u>	<u>setup_dma()</u>
	<u>gei_get_count()</u>	<u>gei_set_count()</u>	<u>setup_gei()</u>
QEI	<u>gei_status()</u>		
	dci_data_received()	dci_read()	<u>dci_start()</u>
DCI			
	<u>dci_transmit_ready()</u>	<u>dci_write()</u>	<u>setup_dci()</u>
	<u>rtc_alarm_read()</u>	<u>rtc_alarm_write()</u>	<u>setup_RTC_alarm()</u>
RTC			
	<u>rtc_read()</u>	<u>rtc_write()</u>	<u>setup_rtc()</u>
CRC	<u>crc_calc(mode)</u>	<u>crc_init(mode)</u>	<u>setup_crc(mode)</u>

[crc_calc8\(\)](#)

[dac_write\(\)](#)

[read_high_speed_adc\(\)](#)

[high_speed_adc_done\(\)](#)

DA CONVERSION

[setup_high_speed_adc_pair\(\)](#)

[setup_dac\(\)](#)

[setup_high_speed_adc\(\)](#)

CAPACITIVE TOUCH

[touchpad_getc\(\)](#)

[touchpad_hit\(\)](#)

[touchpad_state\(\)](#)

EXTENDED RAM

[write_extended_ram\(\)](#)

[read_extended_ram\(\)](#)

[pmp_address\(address\)](#)

[psp_read\(\)](#)

[psp_overflow\(\)](#)

[pmp_overflow\(\)](#)

[setup_pmp\(option,
address_mask\)](#)

[psp_write\(\)](#)

PARALLEL PORT

[pmp_output_full\(\)](#)

[pmp_input_full\(\)](#)

[setup_psp\(option,
address_mask\)](#)

[pmp_write\(\)](#)

[pmp_read\(\)](#)

[psp_input_full\(\)](#)

[psp_output_full\(\)](#)

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 Files:	None
Also See:	setup_adc() , set_adc_channel() , setup_adc_ports() , read_adc() , ADC Overview

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 [ex_tank.c](#)

Files:

Also See: [log\(\)](#), [log10\(\)](#), [exp\(\)](#), [pow\(\)](#), [sqrt\(\)](#)

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 None

Files:

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

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 [ex_tank.c](#)

Files:

Also See: [log\(\)](#), [log10\(\)](#), [exp\(\)](#), [pow\(\)](#), [sqrt\(\)](#)

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 [ex_tank.c](#)

Files:

Also See: [log\(\)](#), [log10\(\)](#), [exp\(\)](#), [pow\(\)](#), [sqrt\(\)](#)

atoe

Syntax: **atoe(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 = atoe(string);
// x is now 12000.00
```

Example
Files:

Also See: [atoi\(\)](#), [atol\(\)](#), [atoi32\(\)](#), [atof\(\)](#), [printf\(\)](#)

atof() atof48() atof64()

Syntax: result = atof (**string**)
or
result = atof48(**string**)
or
result=atof64(**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() atoi48() atoi64()

Syntax:

```
    ivalue = atoi(string)
        or
    lvalue = atol(string)
        or
    i32value = atoi32(string)
        or
    i48value=atoi48(string)
        or
    i64value=atoi64(string)
```

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

Returns:

- ivalue is an 8 bit int.
- lvalue is a 16 bit int.
- i32value is a 32 bit int.
- 48value is a 48 bit int.
- i64value is a 64 bit int.

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 [input.c](#)

Files:

Also See: [printf\(\)](#)

atoi() atol() atoi32() atoi48() atoi64()

Syntax:

```
    ivalue = atoi(string)
        or
    lvalue = atol(string)
        or
```

i32value = atoi32(**string**)
or
i48value=atoi48(**string**)
or
i64value=atoi64(**string**)

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

Returns: ivalue is an 8 bit int.
lvalue is a 16 bit int.
i32value is a 32 bit int.
48value is a 48 bit int.
i64value is a 64 bit int.

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 [input.c](#)

Files:

Also See: [printf\(\)](#)

atoi() atol() atoi32() atoi48() atoi64()

Syntax: ivalue = atoi(**string**)
or
lvalue = atol(**string**)
or
i32value = atoi32(**string**)
or
i48value=atoi48(**string**)
or
i64value=atoi64(**string**)

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

Returns: ivalue is an 8 bit int.
lvalue is a 16 bit int.
i32value is a 32 bit int.

48value is a 48 bit int.
i64value is a 64 bit int.

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\(\)](#)

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_patq.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 Files: [ex_patq.c](#)

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 <stdlib.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:

Also See: [enable_interrupts](#) , [#INT](#) , [Interrupts Overview](#)
[disable_interrupts\(\)](#), [interrupt_active\(\)](#)

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 [ex_tank.c](#)

Files:

Also See: [log\(\)](#), [log10\(\)](#), [exp\(\)](#), [pow\(\)](#), [sqrt\(\)](#)

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 [ex_tank.c](#)

Files:

Also See: [log\(\)](#), [log10\(\)](#), [exp\(\)](#), [pow\(\)](#), [sqrt\(\)](#)

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 the `crc_calc32()`, `crc_calc16()`, or `crc_calc8()` is used.

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 `crc_calc32()`, `crc_calc16()`, or `crc_calc8()` function calls.

width- optional parameter used to specify the input data bit width to use with the `crc_calc32()`, `crc_calc16()`, and `crc_calc8()` functions. 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 None

Files:

Also See: [setup_crc\(\)](#); [crc_init\(\)](#)

crc_init(mode)

Syntax: `crc_init (data);`

Parameters: **data** - This will setup the initial value used by write CRC shift register. Most commonly, this register is set to 0x0000 for start of a new CRC calculation.

Returns: undefined

Function: Configures the CRCWDAT register with the initial value used for CRC calculations.

Availability: Only the devices with built in CRC module.

Requires: Nothing

Examples: `crc_init () ; // Starts the CRC accumulator out at 0
crc_init(0xFFFF) ; // Starts the CRC accumulator out at 0xFFFF`

Example None

Files:

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: `if(cwg_status() == CWG_AUTO_SHUTDOWN)
cwg_restart();`

Example Files:	None
Also See:	setup_cwg(), cwg_restart()

cwg_restart()

Syntax: `cwg_restart();`

Parameters: None

Returns: Nothing

Function: To restart the CWG module after an auto-shutdown event occurs, when not using auto-raster option of module.

Availability: On devices with a CWG module.

Examples: `if(cwg_status() == CWG_AUTO_SHUTDOWN)
cwg_restart();`

Example Files:

Also See: [setup_cwg\(\), cwg_status\(\)](#)

dac_write()

Syntax: `dac_write (value)`
`dac_write (channel, value)`

Parameters: **Value:** 8-bit integer value to be written to the DAC module

Value: 16-bit integer value to be written to the DAC module

channel: Channel to be written to. Constants are:

- DAC_RIGHT
- 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: `int i = 0;`
`setup_dac(DAC_VDD | DAC_OUTPUT);`
`while(1) {`

```

        i++;
        dac_write(i);
    }
    int i = 0;
    setup_dac(DAC_RIGHT_ON, 5);
    while(1) {
        i++;
        dac_write(DAC_RIGHT | i);
    }
}

```

Also See: [setup_dac\(\)](#), [DAC Overview](#), see header file for device selected

dci_read()

Syntax: `dci_read(left_channel, right_channel);`

Parameters: ***left_channel***- A pointer to a signed int16 that will hold the incoming audio data for the left channel (on a stereo system). This data is received on the bus before the right channel data (for situations where left & right channel does have meaning)
right_channel- A pointer to a signed int16 that will hold the incoming audio data for the right channel (on a stereo system). This data is received on the bus after the data in *left channel*.

Returns: undefined

Function: Use this function to read two data words. Do not use this function with DMA. This function is provided mainly for applications involving a stereo codec.

If your application does not use both channels but only receives on a slot (see `setup_dci`), use only the left channel.

Availability: Only available on devices with DCI

Requires: None

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

Example Files:

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 None

Files:

Also See: [DCI Overview](#), [setup_dci\(\)](#), [dci_write\(\)](#), [dci_read\(\)](#), [dci_transmit_ready\(\)](#), [dci_data_received\(\)](#)

dci_transmit_ready()

Syntax: `dci_transmit_ready()`

Parameters: None

Returns: An int1. Returns true if the DCI module is ready to transmit (there is space open in the hardware buffer).

Function: Use this function to poll the transmit buffers.

Availability: Only available on devices with DCI

Requires: None

Examples: `while(1)
{
 if(dci_transmit_ready())
 {
 //transmit data, load buffers, etc...
 }
}`

```
}
```

Example None
Files:

Also See: [DCI Overview](#), [setup_dci\(\)](#), [dci_start\(\)](#), [dci_write\(\)](#), [dci_read\(\)](#), [dci_data_received\(\)](#)

dci_write()

Syntax: `dci_write(left_channel, right_channel);`

Parameters: *left channel*- A pointer to a signed int16 that holds the outgoing audio data for the left channel (on a stereo system). This data is transmitted on the bus before the right channel data (for situations where left & right channel does have meaning)

right channel- A pointer to a signed int16 that holds the outgoing audio data for the right channel (on a stereo system). This data is transmitted on the bus after the data in *left channel*.

Returns: undefined

Function: Use this function to transmit two data words. Do not use this function with DMA. This function is provided mainly for applications involving a stereo codec.

If your application does not use both channels but only transmits on a slot (see `setup_dci()`), use only the left channel. If you transmit more than two slots, call this function multiple times.

Availability: Only available on devices with DCI

Requires: None

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

Example None
Files:

Also See: [DCI Overview](#), [setup_dci\(\)](#), [dci_start\(\)](#), [dci_read\(\)](#), [dci_transmit_ready\(\)](#), [dci_data_received\(\)](#)

delay_cycles()

Syntax: `delay_cycles (count)`

Parameters: *count* - a constant 1-255

Returns: undefined

Function: Creates code to perform a delay of the specified number of instruction clocks (1-255). An instruction clock is equal to four oscillator clocks.

The delay time may be longer than requested if an interrupt is serviced during the delay. The time spent in the ISR does not count toward the delay time.

Availability: All devices

Requires: Nothing

Examples: `delay_cycles(1); // Same as a NOP`
`delay_cycles(25); // At 20 mhz a 5us delay`

Example [ex_cust.c](#)

Files:

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(**Inum**, **Idenom**)

Parameters: **num** and **denom** are signed integers.

num is the numerator and **denom** is the denominator.

Inum and **Idenom** are signed longs , signed int32, int48 or int64

Inum is the numerator and **Idenom** 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 $\text{quot} * \text{denom}(\text{l denom}) + \text{rem}$ shall equal $\text{num}(\text{l num})$.

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 None

Files:

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.

Returns: void

Function: Starts the DMA transfer for the specified channel in the specified mode of operation.

Availability: Devices that have the DMA module.

Requires: Nothing

Examples: dma_start(2, DMA_CONTINOUS | DMA_PING_PONG, 0x4000, 0x4200, 255);

```
// This will setup the DMA channel 2 for continuous ping-pong mode  
with DMA RAM addresses of 0x4000 and 0x4200.
```

Example Files: None
Also See: [setup_dma\(\)](#), [dma_status\(\)](#)

dma_status()

Syntax: 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(INT_CN_PIN|Pin_B0);`

Example Files: None

Also See: [disable_interrupts\(\)](#), [#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 None

Files:

Also See: [write program memory\(\)](#), [Program Eeprom Overview](#)

exp()

Syntax: `result = exp (value)`

Parameters: **value** is any float type

Returns: A float with a precision equal to **value**

Function: Computes the exponential function of the argument. This is e to the power of value where e is the base of natural logarithms. `exp(1)` is 2.7182818.

Note on error handling:

If "errno.h" is included then the domain and range errors are stored in the `errno` variable. The user can check the `errno` to see if an error has occurred and print the error using the `perror` function.

Range error occur in the following case:

- `exp`: when the argument is too large

Availability: All devices

Requires: `#INCLUDE <math.h>`

Examples: `// Calculate x to the power of y`
`x_power_y = exp(y * log(x));`

Example Files:

Also See: [pow\(\)](#), [log\(\)](#), [log10\(\)](#)

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 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 (PCM and PCH)

Requires: Constants are in the devices .h file

Examples: `ext_int_edge(2, L_TO_H); // Set up PIC18 EXT2`
`ext_int_edge(H_TO_L); // Sets up EXT`

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: `double result;`
`result=fabs(-40.0)`
`// result is 40.0`

Example Files: None

Also See: [abs\(\)](#), [labs\(\)](#)

getc() getch() getchar() fgetc()

Syntax: `value = getc()`
`value = fgetc(stream)`
`value=getch()`
`value=getchar()`

Parameters: **stream** is a stream identifier (a constant byte)

Returns: An 8 bit character

Function: This function waits for a character to come in over the RS232 RCV pin and returns

the character. If you do not want to hang forever waiting for an incoming character use kbhit() to test for a character available. If a built-in USART is used the hardware can buffer 3 characters otherwise GETC must be active while the character is being received by the PIC®.

If fgetc() is used then the specified stream is used where getc() defaults to STDIN (the last USE RS232).

Availability: All devices

Requires: #USE RS232

Examples:

```
printf("Continue (Y,N) ?");
do {
    answer=getch();
}while(answer!='Y' && answer!='N');

#use rs232(baud=9600,xmit=pin_c6,
           rcv=pin_c7,stream=HOSTPC)
#use rs232(baud=1200,xmit=pin_b1,
           rcv=pin_b0,stream=GPS)
#use rs232(baud=9600,xmit=pin_b3,
           stream=DEBUG)
...
while(TRUE) {
    c=fgetc(GPS);
    fputc(c,HOSTPC);
    if(c==13)
        fprintf(DEBUG,"Got a CR\r\n");
}
```

Example Files: [ex_stwt.c](#)

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

gets() fgets()

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

Parameters: **string** is a pointer to an array of characters.
Stream is a stream identifier (a constant byte)

Returns: undefined

Function: Reads characters (using getc()) into the string until a RETURN (value 13) is encountered. The string is terminated with a 0. Note that INPUT.C has a more versatile get_string function.

If fgets() is used then the specified stream is used where gets() defaults to STDIN (the last USE RS232).

Availability: All devices

Requires: #USE RS232

Examples:

```
char string[30];  
  
printf("Password: ");  
gets(string);  
if(strcmp(string, password))  
    printf("OK");
```

Example Files: None

Also See: [getc\(\)](#), [get_string](#) in [input.c](#)

floor()

Syntax: 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:

```
// Find the fractional part of a value  
  
frac = value - floor(value);
```

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:

```
float result;
result=fmod(3,2);
// result is 1
```

Example Files:

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](#)

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:	<pre>putc('*'); for(i=0; i<10; i++) putc(buffer[i]); putc(13);</pre>
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:	<pre>puts(" ----- "); puts(" HI "); puts(" ----- ");</pre>

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 <stdlib.h>

Examples:

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

Example Files:	None
Also See:	realloc() , malloc() , calloc()

fexp()

Syntax: result=fexp (*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 fexp 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:

Also See: [ldexp\(\)](#), [exp\(\)](#), [log\(\)](#), [log10\(\)](#), [modf\(\)](#)

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 the current capture values in the result buffer are cleared, an the next result to be sent to the buffer is returned. If **wait** is false, the default setting, the first value currently in the buffer is returned. However, the buffer will only hold four results while waiting for them to be read, so if read isn't being called for every capture event, when **wait** is false, the buffer will fill with old capture values and any new results will be lost.

Availability: Only available on devices with Input Capture modules

Requires: None

Examples:

```
setup_timer3(TMR_INTERNAL | TMR_DIV_BY_8);
setup_capture(2, CAPTURE_FE | CAPTURE_TIMER3);
while(TRUE) {
    timerValue = get_capture(2, TRUE);
    printf("Capture 2 occurred at: %LU", timerValue);
}
```

Example Files:

Also See: [setup_capture\(\)](#), [setup_compare\(\)](#), [Input Capture Overview](#)

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,FASTEAST)
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,FASTEAST)
result = get_capture_time();

Example Files: **None**

Also See: [#use_capture](#), [get_capture_event\(\)](#)

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:

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: value =get_nco_inc_value();

Parameters: None

Returns: - current value set in increment registers.

Availability: On devices with a NCO module.

Examples: value = get_nco_inc_value();

Example
Files: None

Also See: setup_nco(), [set_nco_inc_value\(\), get_nco_accumulator\(\)](#)

get_ticks()

Syntax: value = get_ticks([**stream**]);

Parameters: **stream** – optional parameter specifying the stream defined in #USE TIMER.

Returns: value – a 8, 16, 32 or 64 bit integer. (int8, int16, int32 or int64)

Function: Returns the current tick value of the tick timer. The size returned depends on the

size of the tick timer.

Availability: All devices.

Requires: #USE TIMER(options)

```
#USE TIMER(TIMER=1,TICK=1ms,BITS=16,NOISR)
```

```
void main(void) {
    unsigned int16 current_tick;
    current_tick = get_ticks();
}
```

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 [ex_stwt.c](#)

Files:

Also See: [Timer Overview](#), [setup_timerX\(\)](#), [get_timerXY\(\)](#), [set_timerX\(\)](#), [set_timerXY\(\)](#)

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](#)

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](#)

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 [ex_stwt.c](#)

Files:

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:fffff	Returns 1 if fuse fffff is enabled
FUSE_VALID:fffff INT:iiiii	Returns 1 if fuse fffff is valid 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.
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
ICx	Returns TRUE if ICx is on this part
OCx	Returns TRUE if OCx is on this part

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);

#if getenv("FUSE_VALID:BROWNOUT")
    #FUSE BROWNOUT
#endif

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

Example Files: None

Also See: None

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 getc()) into the string until a RETURN (value 13) is encountered. The string is terminated with a 0. Note that INPUT.C has a more versatile get_string function.

If fgets() is used then the specified stream is used where gets() defaults to STDIN (the last USE RS232).

Availability: All devices

Requires: #USE RS232

Examples:

```
char string[30];

printf("Password: ");
gets(string);
if(strcmp(string, password))
    printf("OK");
```

Example None

Files:

Also See: [getc\(\)](#), [get_string](#) in [input.c](#)

goto_address()

Syntax: `goto_address(location);`

Parameters: *location* is a ROM address, 16 or 32 bit int.

Returns: Nothing

Function: This function jumps to the address specified by *location*. Jumps outside of the current function should be done only with great caution. This is not a normally used function except in very special situations.

Availability: All devices

Requires: Nothing

Examples:

```
#define LOAD_REQUEST PIN_B1
#define LOADER 0x1f00

if(input(LOAD_REQUEST))
    goto_address(LOADER);
```

Example [setjmp.h](#)

Files:

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 None

Files:

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 disabled. 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 None

Files:

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();
    i2c_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 [ex_slave.c](#)

Files:

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: `I2C_Speed (400000);`

Example Files: none

Also See: [i2c_poll](#), [i2c_start](#), [i2c_stop](#), [i2c_slaveaddr](#), [i2c_isr_state](#), [i2c_write](#), [i2c_read](#), [#USE I2C](#), [I2C Overview](#)

i2c_start()

Syntax: `i2c_start()`
`i2c_start(stream)`
`i2c_start(stream, restart)`

Parameters: **stream**: specify the stream defined in #USE I2C

restart: 2 – new restart is forced instead of start
1 – normal start is performed
0 (or not specified) – restart is done only if the compiler last encountered a I2C_START and no I2C_STOP

Returns: undefined

Function: Issues a start condition when in the I2C master mode. After the start condition the clock is held low until I2C_WRITE() is called. If another I2C_start is called in the same function before an i2c_stop is called, then a special restart condition is issued. Note that specific I2C protocol depends on the slave device. The I2C_START function will now accept an optional parameter. If 1 the compiler assumes the bus is in the stopped state. If 2 the compiler treats this I2C_START as a restart. If no parameter is passed a 2 is used only if the compiler compiled a I2C_START last with no I2C_STOP since.

Availability: All devices.

Requires: #USE I2C

Examples:

```
i2c_start();
i2c_write(0xa0);      // Device address
i2c_write(address);  // Data to device
i2c_start();          // Restart
i2c_write(0xa1);      // to change data direction
data=i2c_read(0);     // Now read from slave
i2c_stop();
```

Example Files: [ex_extee.c](#) with [2416.c](#)

Also See: [i2c_poll](#), [i2c_speed](#), [i2c_stop](#), [i2c_slaveaddr](#), [i2c_isr_state](#), [i2c_write](#), [i2c_read](#), [#USE I2C](#), [I2C Overview](#)

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:

```
i2c_start();      // Start condition
i2c_write(0xa0);  // Device address
i2c_write(5);     // Device command
i2c_write(12);    // Device data
```

```
i2c_stop();           // Stop condition
```

Example Files: [ex_extee.c](#) with [2416.c](#)

Also See: [i2c_poll](#), [i2c_speed](#), [i2c_start](#), [i2c_slaveaddr](#), [i2c_isr_state](#), [i2c_write](#), [i2c_read](#),
[#USE I2C](#), [I2C Overview](#)

i2c_write()

Syntax: `i2c_write (data)`
`i2c_write (stream, data)`

Parameters: **data** is an 8 bit int
stream - 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:

```
long cmd;  
...  
i2c_start();      // Start condition  
i2c_write(0xa0); // Device address  
i2c_write(cmd); // Low byte of command  
i2c_write(cmd>>8); // High byte of command  
i2c_stop();       // Stop condition
```

Example Files: [ex_extee.c](#) with [2416.c](#)

Also See: [i2c_poll](#), [i2c_speed](#), [i2c_start](#), [i2c_stop](#), [i2c_slaveaddr](#), [i2c_isr_state](#), [i2c_read](#),
[#USE I2C](#), [I2C Overview](#)

input()

Syntax: `value = input (pin)`

Parameters: **Pin** to read. Pins are defined in the devices .h file. The actual value is a bit

address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651 . This is defined as follows: #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:

```
while ( !input(PIN_B1) );
// waits for B1 to go high

if( input(PIN_A0) )
    printf("A0 is now high\r\n");

int16 i=PIN_B1;
while(!i);
//waits for B1 to go high
```

Example [ex_pulse.c](#)

Files:

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:

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:

Also See: [input\(\)](#), [set_tris_x\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [General Purpose I/O](#)

input_x()

Syntax: `value = input_a()`
`value = input_b()`
`value = input_c()`
`value = input_d()`
`value = input_e()`
`value = input_f()`
`value = input_g()`

```
value = input_h()  
value = input_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: [disable_interrupts\(\)](#) , [#INT](#) , [Interrupts Overview](#)
[clear_interrupt](#), [enable_interrupts\(\)](#)

**isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char)
isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)**

Syntax:

```
value = isalnum(datac)
value = isalpha(datac)
value = isdigit(datac)
value = islower(datac)
value = isspace(datac)
value = isupper(datac)
value = isxdigit(datac)
value = iscntrl(datac)
value = isgraph(datac)
value = isprint(datac)
value = punct(datac)
```

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function: Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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\(\)](#)

**isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char)
isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)**

Syntax: value = isalnum(**datac**)

```
value = isalpha(datac)
value = isdigit(datac)
value = islower(datac)
value = isspace(datac)
value = isupper(datac)
value = isxdigit(datac)
value = iscntrl(datac)
value = isgraph(datac)
value = isprint(datac)
value = punct(datac)
```

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function: Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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 sting

Returns: 0 (or FALSE) if value is not in cstring
1 (or TRUE) if value is in cstring

Function: Returns TRUE if a character is one of the characters in a constant string.

Availability: All devices

Requires: Nothing

Examples:

```
char x= 'x';
...
if ( isamong ( x,
    "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" ) )
printf ("The character is valid");
```

Example Files: #INCLUDE <ctype.h>

Also See: [isalnum\(\)](#), [isalpha\(\)](#), [isdigit\(\)](#), [isspace\(\)](#), [islower\(\)](#), [isupper\(\)](#), [isxdigit\(\)](#)

isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)

Syntax:

```
value = isalnum(datac)
value = isalpha(datac)
value = isdigit(datac)
value = islower(datac)
value = isspace(datac)
value = isupper(datac)
value = isxdigit(datac)
value = iscntrl(datac)
value = isgraph(datac)
value = isprint(datac)
value = punct(datac)
```

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function: Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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\(\)](#)

isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)

Syntax:

```
value = isalnum(datac)
value = isalpha(datac)
value = isdigit(datac)
value = islower(datac)
value = isspace(datac)
value = isupper(datac)
value = isxdigit(datac)
value = iscntrl(datac)
value = isgraph(datac)
value = isprint(datac)
value = punct(datac)
```

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function: Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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 [ex_str.c](#)

Files:

Also See: [isamong\(\)](#)

isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)

Syntax:

```
value = isalnum(datac)
value = isalpha(datac)
value = isdigit(datac)
value = islower(datac)
value = isspace(datac)
value = isupper(datac)
value = isxdigit(datac)
value = iscntrl(datac)
value = isgraph(datac)
value = isprint(datac)
value = punct(datac)
```

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function: Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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\(\)](#)

isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)

Syntax:

```
value = isalnum(datac)
value = isalpha(datac)
value = isdigit(datac)
value = islower(datac)
value = isspace(datac)
value = isupper(datac)
value = isxdigit(datac)
value = iscntrl(datac)
value = isgraph(datac)
value = isprint(datac)
value = punct(datac)
```

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function: Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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\(\)](#)

isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)

Syntax:

```
value = isalnum(datac)
value = isalpha(datac)
value = isdigit(datac)
value = islower(datac)
value = isspace(datac)
value = isupper(datac)
value = isxdigit(datac)
value = iscntrl(datac)
value = isgraph(datac)
value = isprint(datac)
value = punct(datac)
```

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function: Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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 [ex_str.c](#)

Files:

Also See: [isamong\(\)](#)

isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)

Syntax:

```
value = isalnum(datac)
value = isalpha(datac)
value = isdigit(datac)
value = islower(datac)
value = isspace(datac)
value = isupper(datac)
value = isxdigit(datac)
value = iscntrl(datac)
value = isgraph(datac)
value = isprint(datac)
value = punct(datac)
```

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function: Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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 [ex_str.c](#)

Files:

Also See: [isamong\(\)](#)

**isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char)
isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)**

Syntax:

```

value = isalnum(datac)
value = isalpha(datac)
value = isdigit(datac)
value = islower(datac)
value = isspace(datac)
value = isupper(datac)
value = isxdigit(datac)
value = iscntrl(datac)
value = isgraph(datac)
value = isprint(datac)
value = punct(datac)

```

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function:

Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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 [ex_str.c](#)

Files:

Also See: [isamong\(\)](#)

isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)

Syntax:

- value = isalnum(**datac**)
- value = isalpha(**datac**)
- value = isdigit(**datac**)
- value = islower(**datac**)
- value = isspace(**datac**)
- value = isupper(**datac**)
- value = isxdigit(**datac**)
- value = iscntrl(**datac**)
- value = isgraph(**datac**)
- value = isprint(**datac**)
- value = punct(**datac**)

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function: Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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\(\)](#)

isalnum(char) isalpha(char) isdigit(char) islower(char) isspace(char) isupper(char) isxdigit(char) iscntrl(x) isgraph(x) isprint(x) ispunct(x)

Syntax:

- value = isalnum(**datac**)
- value = isalpha(**datac**)
- value = isdigit(**datac**)
- value = islower(**datac**)
- value = isspace(**datac**)
- value = isupper(**datac**)
- value = isxdigit(**datac**)
- value = iscntrl(**datac**)
- value = isgraph(**datac**)
- value = isprint(**datac**)
- value = punct(**datac**)

Parameters: **datac** is a 8 bit character

Returns: 0 (or FALSE) if datac dose not match the criteria, 1 (or TRUE) if datac does match the criteria.

Function: Tests a character to see if it meets specific criteria as follows:

isalnum(x)	X is 0..9, 'A'..'Z', or 'a'..'z'
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\(\)](#)

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 [ex_tgetc.c](#)

Files:

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\(\)](#)

Idexp()

Syntax: `result= Idexp (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 Idexp function multiplies a floating-point number by an integral power of 2.

Availability: All devices.

Requires: #INCLUDE <math.h>

Examples:

```
float result;
result=ldexp(.5,0);
// result is .5
```

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: Computes the natural logarithm of the float x. If the argument is less than or equal to zero or too large, the behavior is undefined.

Note on error handling:

"errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Domain error occurs in the following cases:

- log: when the argument is negative

Availability: All devices

Requires: #INCLUDE <math.h>

Examples: `lnx = log(x);`

Example
Files:

Also See: [log10\(\)](#), [exp\(\)](#), [pow\(\)](#)

log10()

Syntax: `result = log10 (value)`

Parameters: **value** is any float type

Returns: A float with precision equal to **value**

Function: Computes the base-ten logarithm of the float x. If the argument is less than or equal to zero or too large, the behavior is undefined.

Note on error handling:

If "errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Domain error occurs in the following cases:

- log10: when the argument is negative

Availability: All devices

Requires: #INCLUDE <math.h>

Examples: db = log10(read_adc() * (5.0/255)) *10;

Example Files: None

Also See: [log\(\)](#), [exp\(\)](#), [pow\(\)](#)

longjmp()

Syntax: longjmp (*env, val*)

Parameters: ***env***: The data object that will be restored by this function

val: The value that the function setjmp will return. If val is 0 then the function setjmp will return 1 instead.

Returns: After longjmp is completed, program execution continues as if the corresponding invocation of the setjmp function had just returned the value specified by val.

Function: Performs the non-local transfer of control.

Availability: All devices

Requires: #INCLUDE <setjmp.h>

Examples: longjmp (jmpbuf, 1);

Example Files: None

Also See: [setjmp\(\)](#)

make8()

Syntax: i8 = 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(varhigh, varlow)`

Parameters: **varhigh** and **varlow** are 8 bit integers.

Returns: A 16 bit integer

Function: Makes a 16 bit number out of two 8 bit numbers. If either parameter is 16 or 32 bits only the lsb is used. Same as: `i16 = (int16)(varhigh&0xff)*0x100+(varlow&0xff)` except it is done with two byte moves.

Availability: All devices

Requires: Nothing

Examples:

```
long x;
int hi,lo;

x = make16(hi,lo);
```

Example Files: [ltc1298.c](#)

Also See: [make8\(\)](#), [make32\(\)](#)

make32()

Syntax: `i32 = MAKE32(var1, var2, var3, var4)`

Parameters: **var1-4** are a 8 or 16 bit integers. **var2-4** are optional.

Returns: A 32 bit integer

Function: Makes a 32 bit number out of any combination of 8 and 16 bit numbers. Note that the number of parameters may be 1 to 4. The msb is first. If the total bits provided is less than 32 then zeros are added at the msb.

Availability: All devices

Requires: Nothing

Examples:

```
int32 x;
int y;
long z;

x = make32(1,2,3,4); // x is 0x01020304

y=0x12;
z=0x4321;

x = make32(y,z); // x is 0x00124321

x = make32(y,y,z); // x is 0x12124321
```

Example [ex_freqc.c](#)

Files:

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 <stdlibm.h>

Examples:

```
int * iptr;
iptr=malloc(10);
// iptr will point to a block of memory of 10 bytes.
```

Example None

Files:

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 None

Files:

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:

```
memset(arrayA, 0, sizeof(arrayA));
memset(arrayB, '?', sizeof(arrayB));
memset(&structA, 0xFF, sizeof(structA));
```

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:

```
float 48 result, integral;
result=modf(123.987,&integral);
// result is .987 and integral is 123.0000
```

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:

val1	val2	prod
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:

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 None

Files:

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;
    int 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

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;  
    int daylight_savings;  
}  
  
x = offsetof(time_structure, sec);  
// x will be 2  
x = offsetofbit(time_structure, sec);  
// x will be 16  
x = offsetof (time_structure,  
            daylight_savings);  
// x will be 3  
x = offsetofbit(time_structure,  
                daylight_savings);  
// x will be 28
```

Example Files: None

Also See: None

output_x()

Syntax: output_a (**value**)
 output_b (**value**)
 output_c (**value**)
 output_d (**value**)
 output_e (**value**)
 output_f (**value**)

`output_g (value)`
`output_h (value)`
`output_j (value)`
`output_k (value)`

Parameters: **value** is a 16 bit int

Returns: undefined

Function: Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

Availability: All devices, however not all devices have all ports (A-E)

Requires: Nothing

Examples: `OUTPUT_B (0xf0);`

Example Files: [ex_patg.c](#)

Also See: [input\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [output_float\(\)](#), [output_bit\(\)](#), [#USE FIXED_IO](#), [#USE FAST_IO](#), [#USE STANDARD_IO](#), [General Purpose I/O](#)

output_x()

Syntax: `output_a (value)`
`output_b (value)`
`output_c (value)`
`output_d (value)`
`output_e (value)`
`output_f (value)`
`output_g (value)`
`output_h (value)`
`output_j (value)`
`output_k (value)`

Parameters: **value** is a 16 bit int

Returns: undefined

Function: Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

Availability: All devices, however not all devices have all ports (A-E)

Requires: Nothing

Examples: `OUTPUT_B (0xf0);`

Example Files: [ex_patg.c](#)

Also See: [input\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [output_float\(\)](#), [output_bit\(\)](#), [#USE FIXED IO](#), [#USE FAST IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

output_x()

Syntax:

```
output_a (value)
output_b (value)
output_c (value)
output_d (value)
output_e (value)
output_f (value)
output_g (value)
output_h (value)
output_j (value)
output_k (value)
```

Parameters: **value** is a 16 bit int

Returns: undefined

Function: Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

Availability: All devices, however not all devices have all ports (A-E)

Requires: Nothing

Examples: OUTPUT_B (0xf0) ;

Example Files: [ex_patg.c](#)

Also See: [input\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [output_float\(\)](#), [output_bit\(\)](#), [#USE FIXED IO](#), [#USE FAST IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

output_x()

Syntax:

```
output_a (value)
output_b (value)
output_c (value)
output_d (value)
output_e (value)
output_f (value)
output_g (value)
output_h (value)
output_j (value)
output_k (value)
```

Parameters: **value** is a 16 bit int

Returns: undefined

Function: Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

Availability: All devices, however not all devices have all ports (A-E)

Requires: Nothing

Examples: OUTPUT_B(0xf0);

Example Files: [ex_patq.c](#)

Also See: [input\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [output_float\(\)](#), [output_bit\(\)](#), [#USE FIXED_IO](#), [#USE FAST_IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

output_x()

Syntax: output_a (**value**)
output_b (**value**)
output_c (**value**)
output_d (**value**)
output_e (**value**)
output_f (**value**)
output_g (**value**)
output_h (**value**)
output_j (**value**)
output_k (**value**)

Parameters: **value** is a 16 bit int

Returns: undefined

Function: Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

Availability: All devices, however not all devices have all ports (A-E)

Requires: Nothing

Examples: OUTPUT_B(0xf0);

Example Files: [ex_patq.c](#)

Also See: [input\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [output_float\(\)](#), [output_bit\(\)](#), [#USE FIXED_IO](#), [#USE FAST_IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

output_x()

Syntax:

```
output_a (value)
output_b (value)
output_c (value)
output_d (value)
output_e (value)
output_f (value)
output_g (value)
output_h (value)
output_j (value)
output_k (value)
```

Parameters: **value** is a 16 bit int

Returns: undefined

Function: Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

Availability: All devices, however not all devices have all ports (A-E)

Requires: Nothing

Examples: OUTPUT_B (0xf0);

Example
Files:

Also See: [input\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [output_float\(\)](#), [output_bit\(\)](#), [#USE FIXED_IO](#),
[#USE FAST_IO](#), [#USE STANDARD_IO](#), [General Purpose I/O](#)

output_x()

Syntax:

```
output_a (value)
output_b (value)
output_c (value)
output_d (value)
output_e (value)
output_f (value)
output_g (value)
output_h (value)
output_j (value)
output_k (value)
```

Parameters: **value** is a 16 bit int

Returns: undefined

Function: Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

Availability: All devices, however not all devices have all ports (A-E)

Requires: Nothing

Examples: OUTPUT_B(0xf0);

Example Files: [ex_patq.c](#)

Also See: [input\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [output_float\(\)](#), [output_bit\(\)](#), [#USE FIXED IO](#), [#USE FAST IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

output_x()

Syntax: output_a (**value**)
output_b (**value**)
output_c (**value**)
output_d (**value**)
output_e (**value**)
output_f (**value**)
output_g (**value**)
output_h (**value**)
output_j (**value**)
output_k (**value**)

Parameters: **value** is a 16 bit int

Returns: undefined

Function: Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

Availability: All devices, however not all devices have all ports (A-E)

Requires: Nothing

Examples: OUTPUT_B(0xf0);

Example Files: [ex_patq.c](#)

Also See: [input\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [output_float\(\)](#), [output_bit\(\)](#), [#USE FIXED IO](#), [#USE FAST IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

output_x()

Syntax: output_a (**value**)
output_b (**value**)

```
output_c (value)
output_d (value)
output_e (value)
output_f (value)
output_g (value)
output_h (value)
output_j (value)
output_k (value)
```

Parameters: **value** is a 16 bit int

Returns: undefined

Function: Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

Availability: All devices, however not all devices have all ports (A-E)

Requires: Nothing

Examples: OUTPUT_B (0xf0);

Example [ex_patg.c](#)

Files:

Also See: [input\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [output_float\(\)](#), [output_bit\(\)](#), [#USE FIXED_IO](#), [#USE FAST_IO](#), [#USE STANDARD_IO](#), [General Purpose I/O](#)

output_x()

Syntax: output_a (**value**)
output_b (**value**)
output_c (**value**)
output_d (**value**)
output_e (**value**)
output_f (**value**)
output_g (**value**)
output_h (**value**)
output_j (**value**)
output_k (**value**)

Parameters: **value** is a 16 bit int

Returns: undefined

Function: Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

Availability: All devices, however not all devices have all ports (A-E)

Requires: Nothing

Examples: OUTPUT_B (0xf0);

Example Files:	ex_patq.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_bit()

Syntax: `output_bit (pin, value)`

Parameters: **Pins** are defined in the devices .h file. The actual number is a bit address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651. This is defined as follows: #define PIN_A3 5651. The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time. **Value** is a 1 or a 0.

Returns: undefined

Function: Outputs the specified value (0 or 1) to the specified I/O pin. The method of setting the direction register is determined by the last #USE *_IO directive.

Availability: All devices.

Requires: Pin constants are defined in the devices .h file

Examples:

```

output_bit( PIN_B0, 0);
// Same as output_low(pin_B0);

output_bit( PIN_B0,input( PIN_B1 ) );
// Make pin B0 the same as B1

output_bit( PIN_B0,shift_left(&data,1,input(PIN_B1)));
// Output the MSB of data to
// B0 and at the same time
// shift B1 into the LSB of data

int16 i=PIN_B0;
output_bit(i,shift_left(&data,1,input(PIN_B1)));
//same as above example, but
//uses a variable instead of a constant

```

Example Files:	ex_extee.c with 9356.c
Also See:	input() , output_low() , output_high() , output_float() , output_x() , #USE FIXED IO , #USE FAST IO , #USE STANDARD IO , General Purpose I/O

output_drive()

Syntax: output_drive(pin)

Parameters: **Pins** are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651 . This is defined as follows: #DEFINE PIN_A3 5651 .

Returns: undefined

Function: Sets the specified pin to the output mode.

Availability: All devices.

Requires: Pin constants are defined in the devices.h file.

Examples: output_drive(pin_A0); // sets pin_A0 to output its value
 output_bit(pin_B0, input(pin_A0)) // makes B0 the same as A0

Example Files:

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:

Also See: [input\(\)](#), [output_low\(\)](#), [output_high\(\)](#), [output_bit\(\)](#), [output_x\(\)](#), [output_drive\(\)](#), [#USE](#)

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:

Also See: [RS232 I/O Overview](#)

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 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:

result = pmp_output_full()	//PMP only
result = pmp_input_full()	//PMP only
result = pmp_overflow()	//PMP only
result = pmp_error()	//EPMP only
result = pmp_timeout()	//EPMP only

Parameters: None

Returns: A 0 (FALSE) or 1 (TRUE)

Function: These functions check the Parallel Port for the indicated conditions and return TRUE or FALSE.

Availability: This function is only available on devices with Parallel Port hardware on chips.

Requires: Nothing.

Examples:

```
while (pmp_output_full()) ;  
pmp_data = command;  
while(!pmp_input_full()) ;  
if ( pmp_overflow() )  
    error = TRUE;  
else  
    data = pmp_data;
```

Example None

Files:

Also See: [setup_pmp\(\)](#), [pmp_write\(\)](#), [pmp_read\(\)](#)

pmp_read()

Syntax:

```
result = pmp_read ( );           //Parallel Master Port  
result = pmp_read8(address);    //Enhanced Parallel Master Port  
result = pmp_read16(address);    //Enhanced Parallel Master Port  
pmp_read8(address,pointer,count); //Enhanced Parallel Master Port  
pmp_read16(address,pointer,count); //Enhanced Parallel Master Port
```

Parameters: **address**- EPMP only, address in EDS memory that is mapped to address from parallel port device to read data from or start reading data from. (All address in EDS memory are word aligned)

pointer- EPMP only, pointer to array to read data to.

count- EPMP only, number of bytes to read. For pmp_read16() number of bytes must be even.

Returns: For pmp_read(), pmp_read8(address) or pmp_read16() an 8 or 16 bit value. For pmp_read8(address,pointer,count) and pmp_read16(address,pointer,count) undefined.

Function: For PMP module, this will read a byte from the next buffer location. For EPMP module, reads one byte/word or count bytes of data from the address mapped to the EDS memory location. The address is used in conjunction with the offset address set with the setup_pmp_cs1() and setup_pmp_cs2() functions to determine which address lines are high or low during the read.

Availability: Only the devices with a built in Parallel Master Port module or an Enhanced Parallel Master Port module.

Requires: Nothing.

Examples:

```
result = pmp_read();           //PMP reads next byte of
                                //data
result = pmp_read8(0x8000);    //EPMP reads byte of data from the
address mapped           //to first address in
                          //EDS memory.
pmp_read16(0x8002,ptr,16);   //EPMP reads 16 bytes of
                                //data and returns to array
                                //pointed to by ptr
                                //starting at address mapped
                                //to address 0x8002 in
                                //EDS memory.
```

Example None

Files:

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

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_cx1() and setup_pmp_cx2() 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:

```
pmp_write( data );                                //Write the data byte to
                                                 //the next buffer location.
pmp_write8(0x8000,data);                         //EPMP writes the data byte to
                                                 //the address mapped to
                                                 //the first location in
                                                 //EDS memory.
pmp_write16(0x8002,ptr,16);                      //EPMP writes 16 bytes of
                                                 //data pointed to by ptr
                                                 //starting at address mapped
                                                 //to address 0x8002 in
                                                 //EDS Memory
```

Example Files:

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:

`port_a_pullups(FALSE);`

Example Files:

Also See:

[input\(\)](#), [input_x\(\)](#), [output_float\(\)](#)

pow() pwr()

Syntax: $f = \text{pow } (x,y)$
 $f = \text{pwr } (x,y)$

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 [ex_admm.c](#), [ex_lcdkb.c](#)

Files:

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 [ex_profile.c](#)

Files:

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:

Also See:

[setup_psp\(\)](#), PSP 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 [ex_psp.c](#)

Files:

Also See: [setup_psp\(\)](#), PSP 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 [ex_psp.c](#)

Files:

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 or 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:

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:

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 [ex_tgetc.c](#)

Files:

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 off moving data from the transmit buffer into the hardware TX buffer.

If using software RS232, only useful if using CTS flow control, it will check if there is data in transmit buffer to send. If there is data it will then check the CTS flow control line, and if at its active state it will clock out the next data byte.

Availability: All devices

Requires: #USE RS232

Examples: #USE_RS232(UART1,BAUD=9600,TRANSMIT_BUFFER=50,NOTXISR)
printf("Testing Transmit Buffer");
while(TRUE){
 putc_send();
}

Example Files:

Also See: [USE_RS232\(\)](#), [RCV_BUFFER_FULL\(\)](#), [TX_BUFFER_FULL\(\)](#),
[TX_BUFFER_BYTES\(\)](#), [GET\(\)](#), [PUTC\(\) RINTF\(\)](#), [SETUP_UART\(\)](#),
[PUTC\(\) SEND](#)

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

[ex_tgetc.c](#)

Files:

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

None

Files:

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:

```
#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
while(TRUE){
    if(kbhit()){
        c = getc();

        if(c=='F')
            pwm_off();
    }
}
```

Example Files: None

Also See: [#use_pwm](#), [pwm_on\(\)](#), [pwm_set_duty_percent\(\)](#), [pwm_set_duty\(\)](#), [pwm_set_frequency\(\)](#)

pwm_on()

Syntax: `pwm_on([stream]);`

Parameters: **stream** – optional parameter specifying the stream defined in #USE PWM.

Returns: Nothing.

Function: To turn on the PWM signal.

Availability: All devices.

Requires: #USE PWM

Examples:

```
#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
while(TRUE) {
    if(kbhit()) {
        c = getc();

        if(c=='O')
            pwm_on();
    }
}
```

Example Files: None

Also See: [#use_pwm](#), [pwm_off\(\)](#), [pwm_set_duty_percent\(\)](#), [pwm_set_duty\(\)](#), [pwm_set_frequency\(\)](#)

pwm_set_duty()

Syntax: `pwm_set_duty([stream],duty);`

Parameters: **stream** – optional parameter specifying the stream defined in #USE PWM.

duty – an int16 constant or variable specifying the new PWM high time.

Returns: Nothing.

Function: To change the duty cycle of the PWM signal. The duty cycle percentage depends on the period of the PWM signal. This function is faster than `pwm_set_duty_percent()`, but requires you to know what the period of the PWM signal is.

Availability: All devices.

Requires: #USE PWM

Examples: #USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)

Example Files: None

Also See: [#use_pwm](#), [pwm_on\(\)](#), [pwm_off\(\)](#), [pwm_set_frequency\(\)](#), [pwm_set_duty_percent\(\)](#)

pwm_set_duty_percent

Syntax: `pwm_set_duty_percent([stream]), percent`

Parameters: **stream** – optional parameter specifying the stream defined in #USE PWM.
percent - an int16 constant or variable ranging from 0 to 1000 specifying the new PWM duty cycle, D is 0% and 1000 is 100.0%.

Returns: Nothing.

Function: To change the duty cycle of the PWM signal. Duty cycle percentage is based off the current frequency/period of the PWM signal.

Availability: All devices.

Requires: #USE PWM

Examples: #USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
pwm_set_duty_percent(500); //set PWM duty cycle to 50%

Example Files: None

Also See: [#use_pwm](#), [pwm_on\(\)](#), [pwm_off\(\)](#), [pwm_set_frequency\(\)](#), [pwm_set_duty\(\)](#)

pwm_set_frequency

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: #USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
pwm_set_frequency(1000); //set PWM frequency to 1kHz

Example Files: None

Also See: [#use_pwm, pwm_on\(\), pwm_off\(\), pwm_set_duty_percent, pwm_set_duty\(\)](#)

qei_get_count()

Syntax: value = qei_get_count([**unit**]);

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: value = qei_get_counter();

Example Files: None

Also See: [setup_qei\(\) , qei_set_count\(\) , qei_status\(\)](#).

qei_set_count()

Syntax: qei_set_count([**unit**], **value**);

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: qei_set_counter(value);

Example Files: None

Also See: [setup_qei\(\) , qei_get_count\(\) , qei_status\(\)](#).

qei_status()

Syntax: status = qei_status([unit]);

Parameters: **status**- The status of the QEI module
unit- Optional unit number, defaults to 1.

Returns: void

Function: Returns the status of the QEI module.

Availability: Devices that have the QEI module.

Requires: Nothing.

Examples: status = qei_status();

Example Files: None

Also See: [setup_qei\(\)](#) , [qei_set_count\(\)](#) , [qei_get_count\(\)](#).

qsort()

Syntax: qsort (base, num, width, compare)

Parameters: **base**: Pointer to array of sort data
num: Number of elements
width: Width of elements
compare: Function that compares two elements

Returns: None

Function: Performs the shell-metzner sort (not the quick sort algorithm). The contents of the array are sorted into ascending order according to a comparison function pointed to by compare.

Availability: All devices

Requires: #INCLUDE <stdlib.h>

Examples:

```
int nums[5]={ 2,3,1,5,4};  
int compar(void *arg1,void *arg2);  
  
void main() {  
    qsort ( nums, 5, sizeof(int), compar);  
}  
  
int compar(void *arg1,void *arg2) {  
    if ( * (int *) arg1 < ( * (int *) arg2) return -1  
    else if ( * (int *) arg1 == ( * (int *) arg2) return 0  
    else return 1;  
}
```

Example [ex_qsort.c](#)

Files:

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 None

Files:

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=100)`
`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\(\)](#)

recv_buffer_full()

Syntax: `value = recv_buffer_full([stream]);`

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

Returns: TRUE if receive buffer is full, FALSE otherwise.

Function: Function to test if the receive buffer is full.

Availability: All devices

Requires: #USE RS232

Examples: #USE_RS232(UART1,BAUD=9600,RECEIVE_BUFFER=100)

```
void main(void) {
    char c;
    if(recv_buffer_full())
        c = getc();
}
```

Example Files: None

Also See: [USE_RS232\(\)](#), [RCV_BUFFER_BYTES\(\)](#), [TX_BUFFER_BYTES\(\)](#),
[TX_BUFFER_FULL\(\)](#), [GETC\(\)](#), [PUTC\(\)](#), [PRINTF\(\)](#), [SETUP_UART\(\)](#),
[PUTC_SEND\(\)](#)

read_adc() read_adc2()

Syntax: `value = read_adc ([mode])`
`value = read_adc2 ([mode])`

Parameters: `mode` is an optional parameter. If used the values may be:
ADC_START_AND_READ (continually takes readings, this is the default)
ADC_START_ONLY (starts the conversion and returns)
ADC_READ_ONLY (reads last conversion result)

Returns: Either a 8 or 16 bit int depending on #DEVICE ADC= directive.

Function: This function will read the digital value from the analog to digital converter. Calls to setup_adc(), setup_adc_ports() and set_adc_channel() should be made sometime before this function is called. The range of the return value depends on number of bits in the chips A/D converter and the setting in the #DEVICE ADC= directive as follows:

#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(ramPtr, n)`

Parameters: *ramPtr* is the destination pointer for the read results
count is an 8 bit integer

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 than 30K of RAM.

Requires: Nothing

Examples: `unsigned int8 data[8];
read_extended_ram(1, 0x0000, data, 8);`

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**. Due to the 24 bit program instruction size on the PCD devices, every fourth byte will be read as 0x00

Availability: Only devices that allow reads from program memory.

Requires: Nothing

Examples: `char buffer[64];
read_external_memory(0x40000, buffer, 64);`

Example Files: None

Also See: [write program memory\(\)](#), [Program Eeprom Overview](#)

read_high_speed_adc()

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

Parameters: **pair** – Optional parameter that determines which ADC pair number to start and/or

read. Valid values are 0 to total number of ADC pairs. 0 starts and/or reads ADC pair AN0 and AN1, 1 starts and/or reads ADC pair AN2 and AN3, etc. If omitted then a global start and/or read will be performed.

mode – Optional parameter, if used the values may be:

- ADC_START_AND_READ (starts conversion and reads result)
- ADC_START_ONLY (starts conversion and returns)
- ADC_READ_ONLY(reads conversion result)

result – Pointer to return ADC conversion too. Parameter is optional, if not used the read_fast_adc() function can only perform a start.

Returns: Undefined

Function: This function is used to start an analog to digital conversion and/or read the digital value when the conversion is complete. Calls to setup_high_speed_adc() and setup_high_speed_adc_pairs() should be made sometime before this function is called.

When using this function to perform an individual start and read or individual start only, the function assumes that the pair's trigger source was set to 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_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**. Due to the 24 bit program instruction size on the PCD devices, every fourth byte will be read as 0x00

Availability: Only devices that allow reads from program memory.

Requires: Nothing

Examples: `char buffer[64];
read_external_memory(0x40000, buffer, 64);`

Example Files:
None

Also See: [write program memory\(\)](#), [Program Eeprom Overview](#)

read_rom_memory()

Syntax: READ_ROM_MEMORY (**address**, **dataptr**, **count**);

Parameters: **address** is 32 bits. The least significant bit should always be 0.
dataptr is a pointer to one or more bytes.
count is a 16 bit integer

Returns: undefined

Function: Reads **count** bytes from program memory at **address** to **dataptr**. Due to the 24 bit program instruction size on the PCD devices, three bytes are read from each address location.

Availability: Only devices that allow reads from program memory.

Requires: Nothing

Examples: `char buffer[64];
read_program_memory(0x40000, buffer, 64);`

Example Files: None

Also See: [write_eeprom\(\)](#), [read_eeprom\(\)](#), [Program eeprom overview](#)

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 <stdlib.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\(\)](#)

reset_cpu()

Syntax: `reset_cpu()`

Parameters: None

Returns: This function never returns

Function: This is a general purpose device reset. It will jump to location 0 on PCB and PCM parts and also reset the registers to power-up state on the PIC18XXX.

Availability: All devices

Requires: Nothing

Examples: `if (checksum!=0)
 reset_cpu();`

Example Files:

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:

Also See: [restart_wdt\(\)](#), [reset_cpu\(\)](#)

restart_wdt()

Syntax: `restart_wdt()`

Parameters: None

Returns:	undefined												
Function:	<p>Restarts the watchdog timer. If the watchdog timer is enabled, this must be called periodically to prevent the processor from resetting.</p> <p>The watchdog timer is used to cause a hardware reset if the software appears to be stuck.</p> <p>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:</p> <table style="margin-left: 40px;"> <tr> <td style="text-align: right;">PCB/PCM</td> <td style="text-align: center;">PCH</td> </tr> <tr> <td>Enable/Disable</td> <td>#fuses</td> </tr> <tr> <td>e</td> <td>setup_wdt()</td> </tr> <tr> <td>Timeout time</td> <td>setup_wdt() #fuses</td> </tr> <tr> <td>restart</td> <td>restart_wdt(restart_wdt(</td> </tr> <tr> <td></td> <td>))</td> </tr> </table>	PCB/PCM	PCH	Enable/Disable	#fuses	e	setup_wdt()	Timeout time	setup_wdt() #fuses	restart	restart_wdt(restart_wdt())
PCB/PCM	PCH												
Enable/Disable	#fuses												
e	setup_wdt()												
Timeout time	setup_wdt() #fuses												
restart	restart_wdt(restart_wdt(
))												
Availability:	All devices												

Requires:	#FUSES
Examples:	<pre>#fuses WDT // PCB/PCM example // See setup_wdt for a // PIC18 example main() { setup_wdt(WDT_2304MS); while (TRUE) { restart_wdt(); perform_activity(); } }</pre>
Example Files:	ex_wdt.c
Also See:	#FUSES , setup_wdt() , WDT or Watch Dog Timer Overview

rotate_left()

Syntax:	rotate_left (address , bytes)
Parameters:	<p>address is a pointer to memory</p> <p>bytes is a count of the number of bytes to work with.</p>
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

```
x = 0x86;  
rotate_left( &x, 1 );  
// x is now 0xd
```

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: `rtc_alarm_write(&datetime);`

Example Files:
None

Also See: [rtc_read\(\)](#), [rtc_alarm_read\(\)](#), [rtc_alarm_write\(\)](#), [setup_RTC_alarm\(\)](#), [rtc_write\(\)](#), [setup_RTC\(\)](#)

rtc_read()

Syntax: `rtc_read(&datetime);`

Parameters: **datetime**- A structure that will contain the values returned by the RTCC module.

Structure used in read and write functions are defined in the device header file as `rtc_time_t`.

Returns: void

Function: Reads the current value of Time and Date from the RTCC module and stores the structure date time.

Availability: Devices that have the RTCC module.

Requires: Nothing.

Examples: `rtc_read(&datetime);`

Example
Files:

Also See: [rtc_read\(\)](#), [rtc_alarm_read\(\)](#), [rtc_alarm_write\(\)](#), [setup_RTC_alarm\(\)](#), [rtc_write\(\)](#), [setup_RTC\(\)](#)

rtc_write()

Syntax: `rtc_write(&datetime);`

Parameters: **datetime**- A structure that will contain the values to be written to the RTCC module.

Structure used in read and write functions are defined in the device header file as `rtc_time_t`.

Returns: `void`

Function: Writes the date and time to the RTCC module as specified in the structure date time.

Availability: Devices that have the RTCC module.

Requires: Nothing.

Examples: `rtc_write(&datetime);`

Example
Files:

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 (expr)`

Parameters: **expr** is a logical expression.

Returns: None

Function: This function can only be used in an RTOS task. This function waits for **expr** 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 **expr** to be true.

Availability: All devices

Requires: #USE RTOS

Examples: rtos_await(kbhit());

Also See: None

rtos_disable()

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

Syntax: rtos_disable (**task**)

Parameters: **task** is the identifier of a function that is being used as an RTOS task.

Returns: None

Function: This function disables a task which causes the task to not execute until enabled by rtos_enable(). All tasks are enabled by default.

Availability: All devices

Requires: #USE RTOS

Examples: rtos_disable(toggle_green)

Also See: [rtos enable\(\)](#)

rtos_enable()

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

Syntax: rtos_enable (**task**)

Parameters: **task** is the identifier of a function that is being used as an RTOS task.

Returns: None

Function: This function enables a task to execute at it's specified rate.

Availability: All devices

Requires: #USE RTOS

Examples: rtos_enable(toggle_green);

Also See: [rtos disable\(\)](#)

[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: **task** is an optional parameter that is the identifier of a function that is being used as an RTOS task

Returns: A 0 (FALSE) or 1 (TRUE)

Function: This function returns TRUE if the specified task took more time to execute than it was allocated. If no task was specified, then it returns TRUE if any task ran over its allotted execution time.

Availability: All devices

Requires: #USE RTOS(statistics)

Examples: `rtos_overrun()`

Also See: None

rtos_run()

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

Syntax: `rtos_run()`

Parameters: None

Returns: None

Function: This function begins the execution of all enabled RTOS tasks. This function controls the execution of the RTOS tasks at the allocated rate for each task. This

function will return only when `rtos_terminate()` is called.

Availability: All devices

Requires: #USE RTOS

Examples: `rtos_run()`

Also See: [rtos terminate\(\)](#)

rtos_signal()

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

Syntax: `rtos_signal (sem)`

Parameters: **sem** is a global variable that represents the current availability of a shared system resource (a semaphore).

Returns: None

Function: This function can only be used by an RTOS task. This function increments **sem** to let waiting tasks know that a shared resource is available for use.

Availability: All devices

Requires: #USE RTOS

Examples: `rtos_signal(uart_use)`

Also See: [rtos wait\(\)](#)

rtos_stats()

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

Syntax: `rtos_stats(task,&stat)`

Parameters: **task** is the identifier of a function that is being used as an RTOS task.

stat is a structure containing the following:

```
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 = (ticks*hns_per_tick)/10  
};
```

Returns: Undefined

Function: This function returns the statistic data for a specified **task**.

Availability: All devices

Requires: #USE RTOS(statistics)

Examples: rtos_stats(echo, &stats)

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 rtos_run() call in the program. (This function causes rtos_run() to return.)

Availability: All devices

Requires: #USE RTOS

Examples: rtos_terminate()

Also See: [rtos_run\(\)](#)

rtos_wait()

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

Syntax: rtos_wait (**sem**)

Parameters: **sem** is a global variable that represents the current availability of a shared system resource (a semaphore).

Returns: None

Function: This function can only be used by an RTOS task. This function waits for **sem** to be greater than 0 (shared resource is available), then decrements **sem** to claim usage of the shared resource and continues the execution of the rest of the code the RTOS task. This function allows other tasks to execute while the task waits for the shared resource to be available.

Availability: All devices

Requires: #USE RTOS

Examples: rtos_wait(uart_use)

Also See: [rtos signal\(\)](#)

rtos_yield()

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

Syntax: rtos_yield()

Parameters: None

Returns: None

Function: This function can only be used in an RTOS task. This function stops the execution of the current task and returns control of the processor to rtos_run(). When the next task executes, it will start its execution on the line of code after the rtos_yield().

Availability: All devices

Requires: #USE RTOS

Examples:

```
void yield(void)
{
    printf("Yielding...\r\n");
    rtos_yield();
    printf("Executing code after yield\r\n");
}
```

Also See: None

set_adc_channel()

set_adc_channel2()

Syntax: set_adc_channel (**chan** [,**neg**])
set_adc_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 [ex_admm.c](#)

Files:

Also See: [read_adc\(\)](#), [setup_adc\(\)](#), [setup_adc_ports\(\)](#), [ADC Overview](#)

set_compare_time()

Syntax: `set_compare_time(x, ocr, [ocrs])`

Parameters: **x** is 1-8 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, then **ocr** signifies the start of the pulse and **ocrs** defines the pulse termination time.

Availability: Only available on devices with output compare modules.

Requires: Nothing

Examples: `// Pin OC1 will be set when timer 2 is equal to 0xF000
setup_timer2(TMR INTERNAL | TIMER DIV BY 8);
set_compare_time(1, 0xF000);
setup_compare(1, COMPARE_SET_ON_MATCH | COMPARE_TIMER2);`

Example None

Files:

Also See: [get_capture\(\)](#), [setup_compare\(\)](#), [output compare](#), PWM Overview

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:	<code>set_motor_pwm_duty(1,0,0x55); // Sets the PWM1 Unit a duty cycle value</code>
Example Files:	None
Also See:	get_motor_pwm_count() , set_motor_pwm_event() , set_motor_unit() , setup_motor_pwm()

[set_motor_pwm_event\(\)](#)

Syntax:	<code>set_motor_pwm_event(pwm,time);</code>
Parameters:	<p>pwm- Defines the pwm module used.</p> <p>time- The value in the special event comparator register used for scheduling other events.</p>
Returns:	<code>void</code>
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:	get_motor_pwm_count() , setup_motor_pwm() , set_motor_unit() , set_motor_pwm_duty() ;

[set_motor_unit\(\)](#)

Syntax:	<code>set_motor_unit(pwm,unit,options, active_deadtime, inactive_deadtime);</code>
Parameters:	<p>pwm- Defines the pwm module used</p> <p>Unit- This will select Unit A or Unit B</p> <p>options- The mode of the power PWM module. See the devices .h file for all options</p> <p>active_deadtime- Set the active deadtime for the unit</p> <p>inactive_deadtime- Set the inactive deadtime for the unit</p>

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\(\)](#)

s

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 None.

Files:

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_rtcc() set_timer0() set_timer1() set_timer2() set_timer3() set_timer4() set_timer5()

Syntax: `set_timer0(value)` or `set_rtcc (value)`
`set_timer1(value)`
`set_timer2(value)`
`set_timer3(value)`
`set_timer4(value)`
`set_timer5(value)`

Parameters: Timers 1 & 5 get a 16 bit int.

Timer 2 and 4 gets an 8 bit int.

Timer 0 (AKA RTCC) gets an 8 bit int except on the PIC18XXX where it needs a 16 bit int.

Timer 3 is 8 bit on PIC16 and 16 bit on PIC18

Returns: undefined

Function: Sets the count value of a real time clock/counter. RTCC and Timer0 are the same. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2...)

Availability: Timer 0 - All devices
Timers 1 & 2 - Most but not all PCM devices
Timer 3 - Only PIC18XXX and some pick devices
Timer 4 - Some PCH devices
Timer 5 - Only PIC18XX31

Requires: Nothing

Examples: // 20 mhz clock, no prescaler, set timer 0
// to overflow in 35us

```
set_timer0(81);           // 256-(.000035/(4/20000000))
```

Example Files: [ex_patg.c](#)

Also See: [set_timer1\(\)](#), [get_timerX\(\)](#) Timer0 Overview, Timer1Overview, Timer2 Overview, Timer5 Overview

set_ticks()

Syntax: `set_ticks([stream],value);`

Parameters: **stream** – optional parameter specifying the stream defined in #USE TIMER
value – a 8, 16, 32 or 64 bit integer, specifying the new value of the tick timer.
(int8, int16, int32 or int64)

Returns: void

Function: Sets the new value of the tick timer. Size passed depends on the size of the tick timer.

Availability: All devices.

Requires: #USE TIMER(options)

Examples: #USE TIMER(TIMER=1,TICK=1ms,BITS=16,NOISR)

```
void main(void) {
    unsigned int16 value = 0x1000;

    set_ticks(value);
}
```

Example Files: None

Also See: [#USE TIMER](#), [get_ticks\(\)](#)

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/(4/20000000))
```

Example none

Files:

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 none

Files:

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:	<pre>if(EventOccured()) set_timer2(0); //reset the timer.</pre>
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_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_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_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 [ex_patg.c](#)
Files:

Also See: [set_timer1\(\)](#), [get_timerX\(\)](#) Timer0 Overview, Timer1Overview, Timer2 Overview,
Timer5 Overview

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 [ex_patg.c](#)
Files:

Also See: [set_timer1\(\)](#), [get_timerX\(\)](#) Timer0 Overview, Timer1Overview, Timer2 Overview, Timer5 Overview

[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_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_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_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 [lcd.c](#)

Files:

Also See: [#USE FAST IO](#), [#USE FIXED IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

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 [lcd.c](#)

Files:

Also See: [#USE FAST IO](#), [#USE FIXED IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

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_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 [lcd.c](#)

Files:

Also See: [#USE FAST IO](#), [#USE FIXED IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

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 [lcd.c](#)

Files:

Also See: [#USE FAST IO](#), [#USE FIXED IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

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 [lcd.c](#)

Files:

Also See: [#USE FAST IO](#), [#USE FIXED IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

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 [lcd.c](#)

Files:

Also See: [#USE FAST IO](#), [#USE FIXED IO](#), [#USE STANDARD IO](#), [General Purpose I/O](#)

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 [lcd.c](#)

Files:

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 [loader.c](#)

Files:

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 None

Files:

Also See: [longjmp\(\)](#)

setup_adc(mode)

setup_adc2(mode)

Syntax: `setup_adc (mode);`
`setup_adc2(mode);`

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

Parameters:

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 [ex_admm.c](#)

Files:

[setup_adc_ports\(\)](#), [set_adc_channel\(\)](#), [read_adc\(\)](#), [#DEVICE](#), [ADC Overview](#),
see header file for device selected

Also See:

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_capture()

Syntax: `setup_capture(x, mode)`

Parameters: **x** is 1-8 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_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 1-8 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:

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 Moduel 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:

```
setup_crc (12, 5);
// CRC Polynomial is X12 + X5 + 1

setup_crc(16, 15, 3, 1);
// CRC Polynomial is X16 + X15 + X3 + X1+ 1
```

Example Files: [ex.c](#)

Also See: [crc_init\(\)](#); [crc_calc\(\)](#); [crc_calc8\(\)](#)

setup_cwg()

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

Parameters: **mode**- the setup of the CWG module. See the device's .h file for all options. Some typical options include:

- CWG_ENABLED
- CWG_DISABLED
- CWG_OUTPUT_B
- CWG_OUTPUT_A

shutdown- the setup for the auto-shutdown feature of CWG module. See the device's .h file for all the options. Some typical options include:

CWG_AUTO_RESTART
CWG_SHUTDOWN_ON)COMP1
CWG_SHUTDOWN_ON_FLT
CWG_SHUTDOWN_ON_CLC2

dead_time_rising- value specifying the dead time between A and B on the rising edge. (0-63)

dead_time_falling- value specifying the dead time between A and B on the falling edge. (0-63)

Returns: undefined

Function: Sets up the CWG module, the auto-shutdown feature of module and the rising and falling dead times of the module.

Availability: All devices with a CWG module.

Examples: `setup_cwg(CWG_ENABLED|CWG_OUTPUT_A|CWG_OUTPUT_B|
CWG_INPUT_PWM1,CWG_SHUTDOWN_ON_FLT,60,30);`

Example Files: None

Also See: [cwg_status\(\)](#), [cwg_restart\(\)](#)

setup_dac()

Syntax: `setup_dac(mode);`
`setup_dac(mode, divisor);`

Parameters: **mode**- The valid options vary depending on the device. See the devices .h file for all options. Some typical options include:

· DAC_OUTPUT

divisor- Divides the provided clock

Returns: undefined

Function: Configures the DAC including reference voltage. Configures the DAC including channel output and clock speed.

Availability: Only the devices with built in digital to analog converter.

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

Examples: `setup_dac(DAC_VDD | DAC_OUTPUT);`
`dac_write(value);`
`setup_dac(DAC_RIGHT_ON, 5);`

Example Files: None

Also See: [dac_write\(\)](#), [DAC Overview](#), See header file for device selected

setup_dci()

Syntax: `setup_dci(configuration, data size, rx config, tx config, sample rate);`

Parameters: *configuration* - 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:

- 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

data size – Specifies the size of frames and words in the transmission:

- 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

rx config- Specifies which words of a given frame the DCI module will receive (commonly used for a multi-channel, shared bus situation)

- RECEIVE_SLOTx: x May be 0 through 15
- RECEIVE_ALL· RECEIVE_NONE

tx config- Specifies which words of a given frame the DCI module will transmit on.

- TRANSMIT_SLOTx: x May be 0 through 15
- TRANSMIT_ALL
- TRANSMIT_NONE

sample rate-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.

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: `dci_initialize((I2S_MODE | DCI_MASTER | DCI_CLOCK_OUTPUT | SAMPLE_RISING_EDGE | UNDERFLOW_LAST | MULTI_DEVICE_BUS), DCI_1WORD_FRAME | DCI_16BIT_WORD | DCI_2WORD_INTERRUPT, RECEIVE_SLOT0 | RECEIVE_SLOT1, TRANSMIT_SLOT0 | TRANSMIT_SLOT1, 44100);`

Example None

Files:

Also See: [DCI Overview](#), [dci start\(\)](#), [dci write\(\)](#), [dci read\(\)](#), [dci transmit ready\(\)](#), [dci data](#)

[received\(\)](#)

setup_dma()

Syntax: `setup_dma(channel, peripheral, mode);`

Parameters: Channel- The channel used in the DMA transfer
peripheral - The peripheral that the DMA wishes to talk to.
mode- This will specify the mode used in the DMA transfer

Returns: `void`

Function: Configures the DMA module to copy data from the specified peripheral to RAM allocated for the DMA channel.

Availability: Devices that have the DMA module.

Requires Nothing

Examples: `setup_dma(2, DMA_IN_SPI1, DMA_BYTE);`
`// This will setup the DMA channel 1 to talk to`
`// SPI1 input buffer.`

Example Files: None

Also See [dma_start\(\)](#), [dma_status\(\)](#)

setup_high_speed_adc()

Syntax: `setup_high_speed_adc (mode);`

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:

Parameters:

- 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 interrupt 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 None

Files:

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 None

Files:

Also See: [setup_high_speed_adc\(\)](#), [read_high_speed_adc\(\)](#), [high_speed_adc_done\(\)](#)

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)`

Parameter **Pwm**- Defines the pwm module used.
S:
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
postscale- 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: `setup_motor_pwm(1,MPWM_FREE_RUN | MPWM_SYNC_OVERRIDES, timebase);`

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: Mode is one of:

- OSC_INTERNAL
- OSC_CRYSTAL
- OSC_CLOCK
- OSC_RC
- OSC_SECONDARY

Target is the target frequency to run the device it.

Source is optional. It specifies the external crystal/oscillator frequency. If omitted the value from the last #USE_DELAY is used. If mode is OSC_INTERNAL, source is an optional tune value for the internal oscillator for PICs that support it. If omitted a tune value of zero will be used.

Divide is optional. For PICs that support it, it specifies the divide ratio for the Display Module Interface Clock. A number from 0 to 64 divides the clock from 1 to 17 increasing in increments of 0.25, a number from 64 to 96 divides the clock from 17 to 33 increasing in increments of 0.5, and a number from 96 to 127 divides the clock from 33 to 64 increasing in increments of 1. If omitted zero will be used for divide by 1.

Returns: None

Function: Configures the oscillator with preset internal and external source configurations. If the device fuses are set and #use delay() is specified, the compiler will configure the oscillator. Use this function for explicit configuration or programming dynamic clock switches. Please consult your target data sheets for valid configurations, especially when using the PLL multiplier, as many frequency range restrictions are specified.

Availability: This function is available on all devices.

Requires: The configuration constants are defined in the device's header file.

Examples: `setup_oscillator(OSC_CRYSTAL, 4000000, 16000000);`
`setup_oscillator(OSC_INTERNAL, 29480000);`

Example Files:

Also See: [setup_wdt\(\)](#), Internal Oscillator Overview

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
//read/write cycle
- PAR_MASTER_MODE_1 //Master Mode 1
- PAR_WAIT4 //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 | PAR_MASTER_MODE_1 | PAR_STOP_IN_IDLE, 0x00FF);` //Sets up Master mode with address
//lines PMA0:PMA7

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_OFF, 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:

```
setup_power_pwm_pins(PWM_OFF, PWM_OFF, PWM_OFF,  
                      PWM_OFF);  
setup_power_pwm_pins(PWM_COMPLEMENTARY,  
                      PWM_COMPLEMENTARY, PWM_OFF, PWM_OFF);
```

Example None

Files:

Also See: [setup_power_pwm\(\)](#),
[set_power_pwm_override\(\)](#),[set_power_pwmX_duty\(\)](#)

setup_psp(*option,address_mask*)

Syntax:

```
setup_psp (options,address_mask);  
setup_psp(options);
```

Parameters: **Option**- The mode of the Parallel slave port. This allows to set the slave port mode, read-write strobe options and other functionality of the PMP/EPMP module. See the devices .h file for all options. Some typical options include:

- PAR_PSP_AUTO_INC
- PAR_CONTINUE_IN_IDLE
- PAR_INTR_ON_RW //Interrupt on read write
- PAR_INC_ADDR //Increment address by 1 every //read/write cycle
- PAR_WAIT4 //4 Tcy Wait for data hold after //strobe

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).

Returns: Undefined.

Function: Configures various options in the PMP/EPMP module. The options are present in the device.h file and they are used to setup the module. The PMP/EPMP module is highly configurable and this function allows users to setup configurations like the Slave mode, Interrupt options, address increment/decrement options, Address enable bits and various strobe and delay options.

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

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

Examples:

```
setup_psp(PAR_PSP_AUTO_INC|  
          PAR_STOP_IN_IDLE,0x00FF );  
//Sets up legacy slave  
//mode with  
//read and write buffers  
//auto increment.
```

Example 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.

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 None

Files:

Also See: [gei_set_count\(\)](#), [gei_get_count\(\)](#), [gei_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 None

Files:

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_spi() setup_spi2()

Syntax: `setup_spi (mode)`
`setup_spi2 (mode)`

Parameters: *mode* may be:

- SPI_MASTER, SPI_SLAVE, SPI_SS_DISABLED
- SPI_L_TO_H, SPI_H_TO_L
- SPI_CLK_DIV_4, SPI_CLK_DIV_16,
- SPI_CLK_DIV_64, SPI_CLK_T2
- SPI_SAMPLE_AT_END, SPI_XMIT_L_TO_H
- SPI_MODE_16B, SPI_XMIT_L_TO_H
- Constants from each group may be or'ed together with |.

Returns: undefined

Function: Configures the hardware SPI™ module.

- SPI_MASTER will configure the module as the bus master
- SPI_SLAVE will configure the module as a slave on the SPI™ bus
- SPI_SS_DISABLED will turn off the slave select pin so the slave module receives any transmission on the bus.
- SPI_X_to_y will specify the clock edge on which to sample and transmit data
- SPI_CLK_DIV_x will specify the divisor used to create the SCK clock from system clock.

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

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

Examples: `setup_spi(SPI_MASTER | SPI_L_TO_H | SPI_DIV_BY_16);`

Example Files: [ex_spi.c](#)

Also See: [spi_write\(\)](#), [spi_read\(\)](#), [spi_data_is_in\(\)](#), [SPI Overview](#)

setup_timerx()

Syntax: `setup_timerX(mode)`
`setup_timerX(mode,period)`

Parameters: Mode is a bit-field comprised of the following configuration constants:

- TMR_DISABLED: Disables the timer operation.
- TMR_INTERNAL: Enables the timer operation using the system clock. Without divisions, the timer will increment on every instruction cycle. On PCD, this is half the oscillator frequency.
- TMR_EXTERNAL: Uses a clock source that is connected to the SOSCI/SOSCO pins
- T1_EXTERNAL_SYNC: Uses a clock source that is connected to the SOSCI/SOSCO pins. The timer will increment on the rising edge of the external clock which is synchronized to the internal clock phases. This mode is available only for Timer1.
- T1_EXTERNAL_RTC: Uses a low power clock source connected to the SOSCI/SOSCO pins; suitable for use as a real time clock. If this mode is used, the low power oscillator will be enabled by the `setup_timer` function. This mode is available only for Timer1.
- TMR_DIV_BY_X: X is the number of input clock cycles to pass before the timer is incremented. X may be 1, 8, 64 or 256.
- TMR_32_BIT: This configuration concatenates the timers into 32 bit mode. This constant should be used with timers 2, 4, 6 and 8 only.
- Period is an optional 16 bit integer parameter that specifies the timer period. The default value is 0xFFFF.

Returns: `void`

Function: Sets up the timer specified by X (May be 1 – 9). X must be a valid timer on the target device.

Availability: This function is available on all devices that have a valid timer X. Use `getenv` or refer to the target datasheet to determine which timers are valid.

Requires: Configuration constants are defined in the device's header file.

Examples:

```
/* setup a timer that increments every 64th instruction cycle with
an overflow period of 0xA010 */
setup_timer2(TMR_INTERNAL | TMR_DIV_BY_64, 0xA010);

/* Setup another timer as a 32-bit hybrid with a period of
0xFFFFFFFF and a interrupt that will be fired when that timer
overflows*/
setup_timer4(TMR_32_BIT); //use get_timer45() to get the timer
value
enable_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:

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: `setup_timer_B(TB_OFF);`

```
setup_timer_B(TB_INTERNAL | TB_DIV_256);  
setup_timer_B(TA_EXT_L_TO_H | TB_DIV_1);
```

Example Files:
Also See: [get_timerB\(\)](#), [set_timerB\(\)](#), [TimerB Overview](#)

setup_timer_0()

Syntax: `setup_timer_0 (mode)`

Parameters: ***mode*** may be one or two of the constants defined in the devices .h file. RTCC_INTERNAL, RTCC_EXT_L_TO_H or RTCC_EXT_H_TO_L
RTCC_DIV_2, RTCC_DIV_4, RTCC_DIV_8, RTCC_DIV_16, RTCC_DIV_32, RTCC_DIV_64, RTCC_DIV_128, RTCC_DIV_256
PIC18XXX only: RTCC_OFF, RTCC_8_BIT
One constant may be used from each group or'ed together with the | operator.

Returns: undefined

Function: Sets up the timer 0 (aka RTCC).

Availability: All devices.

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

Examples: `setup_timer_0 (RTCC_DIV_2|RTCC_EXT_L_TO_H);`

Example Files:
Also See: [get_timer0\(\)](#), [set_timer0\(\)](#), [setup counters\(\)](#)

setup_timer_1()

Syntax: `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, and 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(). Timer 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 308.8us.
```

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 None

Files:

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 None

Files:

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:

Also See: [#USE RS232](#), [putc\(\)](#), [getc\(\)](#), [RS232 I/O Overview](#)

setup_vref()

Syntax: `setup_vref (mode)`

Parameters: **mode** is a bit-field comprised of the following constants:

- VREF_DISABLED
- VREF_LOW ($Vdd * value / 24$)
- VREF_HIGH ($Vdd * value / 32 + Vdd/4$)
- VREF_ANALOG

Returns: undefined

Function: Configures the voltage reference circuit used by the voltage comparator.

The voltage reference circuit allows you to specify a reference voltage that the comparator module may use. You may use the Vdd and Vss voltages as your reference or you may specify VREF_ANALOG to use supplied Vdd and Vss. Voltages may also be tuned to specific values in steps, 0 through 15. That value must be or'ed to the configuration constants.

Availability: Some devices, consult your target datasheet.

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

Examples: `/* Use the 15th step on the coarse setting */`
`setup_vref(VREF_LOW | 14);`

Example Files:

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](#)

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 Files: [ex_extee.c, 9356.c](#)

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 Files: [ex_extee.c](#), [9356.c](#)

Also See: [shift_left\(\)](#), [rotate_right\(\)](#), [rotate_left\(\)](#),

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\(\)](#)

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 [ex_tank.c](#)

Files:

Also See: [log\(\)](#), [log10\(\)](#), [exp\(\)](#), [pow\(\)](#), [sqrt\(\)](#)

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 [ex_wakup.c](#)

Files:

Also See: [reset cpu\(\)](#)

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: (!spi_data_is_in() && input(PIN_B2)) ;
 if(spi_data_is_in())
 data = spi_read();

Example Files: None

Also See: [spi_read\(\)](#), [spi_write\(\)](#), [SPI Overview](#)

spi_init()

Syntax: spi_init(baud);
 spi_init(stream,baud);

Parameters: **stream** – is the SPI stream to use as defined in the STREAM=name option in #USE SPI.
band- the band rate to initialize the SPI module to. If FALSE it will disable the SPI module, if TRUE it will enable the SPI module to the band rate specified in #use SPI.

Returns: Nothing.

Function: Initializes the SPI module to the settings specified in #USE SPI.

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

Requires: #USE SPI

Examples: #use spi(MATER, SPI1, baud=1000000, mode=0, stream=SPI1_MODE0)

 spi_init(SPI1_MODE0, TRUE); //initialize and enable SPI1 to
 setting in #USE SPI
 spi_init(FALSE); //disable SPI1
 spi_init(250000); //initialize and enable SPI1 to a baud rate of
 250K

Example
Files: None

Also See: #USE SPI, [spi_xfer\(\)](#), [spi_xfer_in\(\)](#), [spi_prewrite\(\)](#), [spi_speed\(\)](#)

spi_prewrite(data);

Syntax: `spi_prewrite(data);`
 `spi_prewrite(stream, data);`

Parameters: **stream** – is the SPI stream to use as defined in the STREAM=name option in #USE SPI.
data- the variable or constant to transfer via SPI

Returns: Nothing.

Function: Writes data into the SPI buffer without waiting for transfer to be completed. Can be used in conjunction with spi_xfer() with no parameters to transfer more 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 `ex_spi_slave.c`

Files:

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 [ex_spi.c](#)

Files:

Also See: [spi_write\(\)](#), [spi_write_16\(\)](#), [spi_read_16\(\)](#), [spi_data_is_in\(\)](#), SPI Overview

[spi_read_16\(\)](#)

[spi_read2_16\(\)](#)

[spi_read3_16\(\)](#)

[spi_read4_16\(\)](#)

Syntax: `value = spi_read_16([data]);`
`value = spi_read2_16([data]);`
`value = spi_read3_16([data]);`
`value = spi_read4_16([data]);`

Parameters: `data` – optional parameter and if included is a 16 bit int

Returns: A 16 bit int

Function: Return a value read by the SPI. If a value is passed to the `spi_read_16()` the data will be clocked out and the data received will be returned. If no data is ready, `spi_read_16()` will wait for the data is a SLAVE or return the last DATA clocked in from `spi_write_16()`.
If this device is the MASTER then either do a `spi_write_16(data)` followed by a `spi_read_16()` or do a `spi_read_16(data)`. These both do the same thing and will generate a clock. If there is no data to send just do a `spi_read_16(0)` to get the clock.
If this device is a slave then either call `spi_read_16()` to wait for the clock and data or use `spi_data_is_in()` to determine if data is ready.

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

Requires: NThat the option SPI_MODE_16B be used in `setup_spi()` function, or that the option XFER16 be used in `#use SPI()`

Examples: `data_in = spi_read_16(out_data);`

Example None

Files:

Also See: [spi_read\(\)](#), [spi_write\(\)](#), [spi_write_16\(\)](#), [spi_data_is_in\(\)](#), SPI Overview

[spi_speed](#)

Syntax: `spi_speed(baud);`
`spi_speed(stream,baud);`
`spi_speed(stream,baud,clock);`

Parameters: **stream** – is the SPI stream to use as defined in the STREAM=name option in #USE SPI.
band- the band rate to set the SPI module to
clock- the current clock rate to calculate the band rate with.
If not specified it uses the value specified in `#use delay()`.

Returns: Nothing.

Function: Sets the SPI module's baud rate to the specified value.

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

Requires: #USE SPI

Examples: spi_speed(250000);
spi_speed(SPI1_MODE0, 250000);
spi_speed(SPI1_MODE0, 125000, 8000000);

Example Files: None

Also See: #USE SPI, spi_xfer(), spi_xfer_in(), spi_prewrite(), spi_init()

[spi_write\(\)](#) [spi_write2\(\)](#)

[spi_write3\(\)](#)

[spi_write4\(\)](#)

Syntax: spi_write([wait],**value**);
spi_write2([wait],**value**);
spi_write3([wait],**value**);
spi_write4([wait],**value**);

Parameters: **value** is an 8 bit int
wait- an optional parameter specifying whether the function will wait for the SPI transfer to complete before exiting. Default is TRUE if not specified.

Returns: Nothing

Function: Sends a byte out the SPI interface. This will cause 8 clocks to be generated. This function will write the value out to the SPI. At the same time data is clocked out data is clocked in and stored in a receive buffer. spi_read() may be used to read the buffer.

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

Requires: Nothing

Examples: spi_write(data_out);
data_in = spi_read();

Example Files: [ex_spi.c](#)

Also See: [spi_read\(\)](#), [spi_data_is_in\(\)](#), [SPI Overview](#), [spi_write_16\(\)](#), [spi_read_16\(\)](#)

[spi_xfer\(\)](#)

Syntax: spi_xfer(data)
spi_xfer(stream, data)
spi_xfer(stream, data, bits)
result = spi_xfer(data)

```
result = spi_xfer(stream, data)
result = spi_xfer(stream, data, bits)
```

Parameters: **data** is the variable or constant to transfer via SPI. The pin used to transfer *data* is defined in the DO=pin option in #use spi. **stream** is the SPI stream to use as defined in the STREAM=name option in #USE SPI.
bits is how many bits of data will be transferred.

Returns: The data read in from the SPI. The pin used to transfer result is defined in the DI=pin option in #USE SPI.

Function: Transfers data to and reads data from an SPI device.

Availability: All devices with SPI support.

Requires: #USE SPI

Examples:

```
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:

```
data_in = spi_xfer_in();
```

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 specifies do not work in ram band strings.

Returns: Bytes is the number of bytes written to string.

Function: This function operates like printf() except that the output is placed into the specified string. The output string will be terminated with a null. No checking is done to ensure the string is large enough for the data. See printf() for details on formatting.

Availability: All devices.

Requires: Nothing

Examples:

```
char mystring[20];
long mylong;

mylong=1234;
sprintf(mystring,"<%lu>",mylong);
// mystring now has:
//      < 1 2 3 4 > \0
```

Example None

Files:

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: Computes the non-negative square root of the float value x. If the argument is negative, the behavior is undefined.

Note on error handling:

If "errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Domain error occurs in the following cases:
sqrt: when the argument is negative

Availability: All devices.

Requires: #INCLUDE <math.h>

Examples: `distance = sqrt(pow((x1-x2),2)+pow((y1-y2),2));`

Example None
Files:
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: `srand(10);
I=rand();`

Example None
Files:
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: <code>ptr=strcat (<i>s1, s2</i>)</code>	Concatenate s2 onto s1
<code>ptr=strchr (<i>s1, c</i>)</code>	Find c in s1 and return &s1[i]
<code>ptr=strrchr (<i>s1, c</i>)</code>	Same but search in reverse
<code>cresult=strcmp (<i>s1, s2</i>)</code>	Compare s1 to s2
<code>iresult=strncmp (<i>s1, s2, n</i>)</code>	Compare s1 to s2 (n bytes)
<code>iresult=stricmp (<i>s1, s2</i>)</code>	Compare and ignore case
<code>ptr=strncpy (<i>s1, s2, n</i>)</code>	Copy up to n characters s2->s1
<code>iresult=strcspn (<i>s1, s2</i>)</code>	Count of initial chars in s1 not in s2
<code>iresult=strspn (<i>s1, s2</i>)</code>	Count of initial chars in s1 also in s2
<code>iresult=strlen (<i>s1</i>)</code>	Number of characters in s1
<code>ptr=strlwr (<i>s1</i>)</code>	Convert string to lower case
<code>ptr=strpbrk (<i>s1, s2</i>)</code>	Search s1 for first char also in s2
<code>ptr=strstr (<i>s1, s2</i>)</code>	Search for s2 in s1
<code>ptr=strncat(<i>s1,s2</i>)</code>	Concatenates up to n bytes of s2 onto s1
<code>iresult=strcoll(<i>s1,s2</i>)</code>	Compares s1 to s2, both interpreted as appropriate to the current locale.

<code>res=strxfrm(<i>s1,s2,n</i>)</code>	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)
<code>iresult=memcmp(<i>m1,m2,n</i>)</code>	Compare m1 to m2 (n bytes)
<code>ptr=memchr(<i>m1,c,n</i>)</code>	Find c in first n characters of m1 and return &m1[i]
<code>ptr=strerror(<i>errnum</i>)</code>	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
 iresult is an 8 bit int
 result is -1 (less than), 0 (equal) or 1 (greater than)
 res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples: `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 [ex_str.c](#)

Files:

Also See: [strcpy\(\)](#), [strtok\(\)](#)

STANDARD STRING FUNCTIONS() memchr() memcmp() strcat() strchr() strcmp() strcoll() strcspn() strerror() stricmp() strlen() strlwr() strncat() strncmp() strncpy() strpbrk() strrchr() strspn() strstr() strxfrm()

Syntax: <code>ptr=strcat (<i>s1, s2</i>)</code>	Concatenate s2 onto s1
<code>ptr=strchr (<i>s1, c</i>)</code>	Find c in s1 and return &s1[i]
<code>ptr=strrchr (<i>s1, c</i>)</code>	Same but search in reverse
<code>cresult=strcmp (<i>s1, s2</i>)</code>	Compare s1 to s2
<code>iresult=strncmp (<i>s1, s2, n</i>)</code>	Compare s1 to s2 (n bytes)
<code>iresult=stricmp (<i>s1, s2</i>)</code>	Compare and ignore case

<code>ptr=strncpy (<i>s1, s2, n</i>)</code>	Copy up to n characters s2->s1
<code>iresult=strcspn (<i>s1, s2</i>)</code>	Count of initial chars in s1 not in s2
<code>iresult=strspn (<i>s1, s2</i>)</code>	Count of initial chars in s1 also in s2
<code>iresult=strlen (<i>s1</i>)</code>	Number of characters in s1
<code>ptr=strlwr (<i>s1</i>)</code>	Convert string to lower case
<code>ptr=strpbrk (<i>s1, s2</i>)</code>	Search s1 for first char also in s2
<code>ptr=strstr (<i>s1, s2</i>)</code>	Search for s2 in s1
<code>ptr=strncat(<i>s1,s2</i>)</code>	Concatenates up to n bytes of s2 onto s1
<code>iresult=strcoll(<i>s1,s2</i>)</code>	Compares s1 to s2, both interpreted as appropriate to the current locale.
<code>res=strxfrm(<i>s1,s2,n</i>)</code>	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)
<code>iresult=memcmp(<i>m1,m2,n</i>)</code>	Compare m1 to m2 (n bytes)
<code>ptr=memchr(<i>m1,c,n</i>)</code>	Find c in first n characters of m1 and return &m1[i]
<code>ptr= strerror(errnum)</code>	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: `ptr` is a copy of the `s1` pointer
 `iresult` is an 8 bit int
 `result` is -1 (less than), 0 (equal) or 1 (greater than)
 `res` is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: `#include <string.h>`

Examples: `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 [ex_str.c](#)

Files:

Also See: `strcpy()`, [strtok\(\)](#)

STANDARD STRING FUNCTIONS()

memchr() **memcmp()** **strcat()**
strchr() **strcmp()** **strcoll()** **strcspn()** **strerror()** **stricmp()** **strlen()**
strlwr() **strncat()** **strncmp()** **strncpy()** **struprbrk()** **strrchr()** **strspn()**
strstr() **strxfrm()**

Syntax:	ptr=strcat (s1, s2) ptr=strchr (s1, c) ptr= strrchr (s1, c) cresult=strcmp (s1, s2) iresult=strncmp (s1, s2, n) iresult=strcmp (s1, s2) ptr= strncpy (s1, s2, n) iresult= strcspn (s1, s2) iresult= strspn (s1, s2) iresult= strlen (s1) ptr= strlwr (s1) ptr=struprbrk (s1, s2) ptr= strstr (s1, s2) ptr= strncat(s1,s2) iresult= strcoll(s1,s2) res= strxfrm(s1,s2,n)	Concatenate s2 onto s1 Find c in s1 and return &s1[i] Same but search in reverse Compare s1 to s2 Compare s1 to s2 (n bytes) Compare and ignore case Copy up to n characters s2->s1 Count of initial chars in s1 not in s2 Count of initial chars in s1 also in s2 Number of characters in s1 Convert string to lower case Search s1 for first char also in s2 Search for s2 in s1 Concatenates up to n bytes of s2 onto s1 Compares s1 to s2, both interpreted as appropriate to the current locale. 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)
	iresult=memcmp(m1,m2,n) ptr=memchr(m1,c,n) ptr= strerror(errnum)	Compare m1 to m2 (n bytes) Find c in first n characters of m1 and return &m1[i] Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
iresult is an 8 bit int
result is -1 (less than), 0 (equal) or 1 (greater than)
res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples: 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 [ex_str.c](#)

Files:

Also See: [strcpy\(\)](#), [strtok\(\)](#)

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=strcmp (s1, c)	Same but search in reverse
	cresult=strcmp (s1, s2)	Compare s1 to s2
	iresult=strncmp (s1, s2, n)	Compare s1 to s2 (n bytes)
	iresult=stricmp (s1, s2)	Compare and ignore case
	ptr=strncpy (s1, s2, n)	Copy up to n characters s2->s1
	iresult=strcspn (s1, s2)	Count of initial chars in s1 not in s2
	iresult=strspn (s1, s2)	Count of initial chars in s1 also in s2
	iresult=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)	Concatenates up to n bytes of s2 onto s1
	iresult=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)
	iresult=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[i]
	ptr=strerror(errnum)	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
 iresult is an 8 bit int
 result is -1 (less than), 0 (equal) or 1 (greater than)
 res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples:

```
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 [ex_str.c](#)

Files:

Also See: [strcpy\(\)](#), [strtok\(\)](#)

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 (<i>s1, s2</i>)	Concatenate s2 onto s1
	ptr=strchr (<i>s1, c</i>)	Find c in s1 and return &s1[i]
	ptr= strrchr (<i>s1, c</i>)	Same but search in reverse
	cresult=strcmp (<i>s1, s2</i>)	Compare s1 to s2
	iresult=strncmp (<i>s1, s2, n</i>)	Compare s1 to s2 (n bytes)
	iresult=stricmp (<i>s1, s2</i>)	Compare and ignore case
	ptr=strncpy (<i>s1, s2, n</i>)	Copy up to n characters s2->s1
	iresult=strcspn (<i>s1, s2</i>)	Count of initial chars in s1 not in s2
	iresult=strspn (<i>s1, s2</i>)	Count of initial chars in s1 also in s2
	iresult=strlen (<i>s1</i>)	Number of characters in s1
	ptr=strlwr (<i>s1</i>)	Convert string to lower case
	ptr=strpbrk (<i>s1, s2</i>)	Search s1 for first char also in s2
	ptr= strstr (<i>s1, s2</i>)	Search for s2 in s1
	ptr=strncat(<i>s1,s2</i>)	Concatenates up to n bytes of s2 onto s1
	iresult= strcoll(<i>s1,s2</i>)	Compares s1 to s2, both interpreted as appropriate to the current locale.
	res=strxfrm(<i>s1,s2,n</i>)	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)
	iresult=memcmp(<i>m1,m2,n</i>)	Compare m1 to m2 (n bytes)
	ptr=memchr(<i>m1,c,n</i>)	Find c in first n characters of m1 and return &m1[i]
	ptr=strerror(errno)	Maps the error number in errno to an error message string. The parameters 'errno' 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
 iresult is an 8 bit int
 result is -1 (less than), 0 (equal) or 1 (greater than)
 res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples: 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 [ex_str.c](#)

Files:

Also See: strcpy(), [strtok\(\)](#)

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
	cresult=strcmp (s1, s2)	Compare s1 to s2
	iresult=strncmp (s1, s2, n)	Compare s1 to s2 (n bytes)
	iresult=stricmp (s1, s2)	Compare and ignore case
	ptr=strncpy (s1, s2, n)	Copy up to n characters s2->s1
	iresult=strcspn (s1, s2)	Count of initial chars in s1 not in s2
	iresult=strspn (s1, s2)	Count of initial chars in s1 also in s2
	iresult=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)	Concatenates up to n bytes of s2 onto s1
	iresult=strcoll(s1,s2)	Compares s1 to s2, both interpreted as appropriate to the current locale.

<code>res=strxfrm(<i>s1,s2,n</i>)</code>	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)
<code>iresult=memcmp(<i>m1,m2,n</i>)</code>	Compare m1 to m2 (n bytes)
<code>ptr=memchr(<i>m1,c,n</i>)</code>	Find c in first n characters of m1 and return &m1[i]
<code>ptr=strerror(<i>errnum</i>)</code>	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
 iresult is an 8 bit int
 result is -1 (less than), 0 (equal) or 1 (greater than)
 res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples:

```
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 [ex_str.c](#)

Files:

Also See: strcpy(), [strtok\(\)](#)

STANDARD STRING FUNCTIONS() memchr() memcmp() strcat() strchr() strcmp() strcoll() strcspn() strerror() stricmp() strlen() strlwr() strncat() strncmp() strncpy() strpbrk() strrchr() strspn() strstr() strxfrm()

Syntax:	<code>ptr=strcat (<i>s1, s2</i>)</code>	Concatenate s2 onto s1
	<code>ptr=strchr (<i>s1, c</i>)</code>	Find c in s1 and return &s1[i]
	<code>ptr=strrchr (<i>s1, c</i>)</code>	Same but search in reverse
	<code>cresult=strcmp (<i>s1, s2</i>)</code>	Compare s1 to s2
	<code>iresult=strncmp (<i>s1, s2, n</i>)</code>	Compare s1 to s2 (n bytes)
	<code>iresult=stricmp (<i>s1, s2</i>)</code>	Compare and ignore case

<code>ptr=strncpy (<i>s1, s2, n</i>)</code>	Copy up to n characters s2->s1
<code>iresult=strcspn (<i>s1, s2</i>)</code>	Count of initial chars in s1 not in s2
<code>iresult=strspn (<i>s1, s2</i>)</code>	Count of initial chars in s1 also in s2
<code>iresult=strlen (<i>s1</i>)</code>	Number of characters in s1
<code>ptr=strlwr (<i>s1</i>)</code>	Convert string to lower case
<code>ptr=strpbrk (<i>s1, s2</i>)</code>	Search s1 for first char also in s2
<code>ptr=strstr (<i>s1, s2</i>)</code>	Search for s2 in s1
<code>ptr=strncat(<i>s1,s2</i>)</code>	Concatenates up to n bytes of s2 onto s1
<code>iresult=strcoll(<i>s1,s2</i>)</code>	Compares s1 to s2, both interpreted as appropriate to the current locale.
<code>res=strxfrm(<i>s1,s2,n</i>)</code>	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)
<code>iresult=memcmp(<i>m1,m2,n</i>)</code>	Compare m1 to m2 (n bytes)
<code>ptr=memchr(<i>m1,c,n</i>)</code>	Find c in first n characters of m1 and return &m1[i]
<code>ptr= strerror(errnum)</code>	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: `ptr` is a copy of the `s1` pointer
 `iresult` is an 8 bit int
 `result` is -1 (less than), 0 (equal) or 1 (greater than)
 `res` is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: `#include <string.h>`

Examples: `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 [ex_str.c](#)

Files:

Also See: `strcpy()`, [strtok\(\)](#)

STANDARD STRING FUNCTIONS()

memchr() **memcmp()** **strcat()**
strchr() **strcmp()** **strcoll()** **strcspn()** **strerror()** **stricmp()** **strlen()**
strlwr() **strncat()** **strcmp()** **strncpy()** **strupr()** **strrchr()** **strspn()**
strstr() **strxfrm()**

Syntax:	ptr=strcat (s1, s2) ptr=strchr (s1, c) ptr=strcmp (s1, c) cresult=strcmp (s1, s2) iresult=strncmp (s1, s2, n) iresult=strcmp (s1, s2) ptr=strncpy (s1, s2, n) iresult=strcspn (s1, s2) iresult=strspn (s1, s2) iresult=strlen (s1) ptr=strlwr (s1) ptr=strupr (s1, s2) ptr=strcmp (s1, s2) ptr=strncat(s1,s2) iresult=strcoll(s1,s2) res=strxfrm(s1,s2,n)	Concatenate s2 onto s1 Find c in s1 and return &s1[i] Same but search in reverse Compare s1 to s2 Compare s1 to s2 (n bytes) Compare and ignore case Copy up to n characters s2->s1 Count of initial chars in s1 not in s2 Count of initial chars in s1 also in s2 Number of characters in s1 Convert string to lower case Search s1 for first char also in s2 Search for s2 in s1 Concatenates up to n bytes of s2 onto s1 Compares s1 to s2, both interpreted as appropriate to the current locale. 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)
	iresult=memcmp(m1,m2,n) ptr=memchr(m1,c,n) ptr=strerror(errnum)	Compare m1 to m2 (n bytes) Find c in first n characters of m1 and return &m1[i] Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
iresult is an 8 bit int
result is -1 (less than), 0 (equal) or 1 (greater than)
res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples: 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 [ex_str.c](#)

Files:

Also See: [strcpy\(\)](#), [strtok\(\)](#)

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=strcmp (s1, c)	Same but search in reverse
	cresult=strcmp (s1, s2)	Compare s1 to s2
	iresult=strncmp (s1, s2, n)	Compare s1 to s2 (n bytes)
	iresult=stricmp (s1, s2)	Compare and ignore case
	ptr=strncpy (s1, s2, n)	Copy up to n characters s2->s1
	iresult=strcspn (s1, s2)	Count of initial chars in s1 not in s2
	iresult=strspn (s1, s2)	Count of initial chars in s1 also in s2
	iresult=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)	Concatenates up to n bytes of s2 onto s1
	iresult=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)
	iresult=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[i]
	ptr=strerror(errnum)	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
 iresult is an 8 bit int
 result is -1 (less than), 0 (equal) or 1 (greater than)
 res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples:

```
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 [ex_str.c](#)

Files:

Also See: [strcpy\(\)](#), [strtok\(\)](#)

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 (<i>s1, s2</i>)	Concatenate s2 onto s1
	ptr=strchr (<i>s1, c</i>)	Find c in s1 and return &s1[i]
	ptr= strrchr (<i>s1, c</i>)	Same but search in reverse
	cresult=strcmp (<i>s1, s2</i>)	Compare s1 to s2
	iresult=strncmp (<i>s1, s2, n</i>)	Compare s1 to s2 (n bytes)
	iresult=strcasecmp (<i>s1, s2</i>)	Compare and ignore case
	ptr=strncpy (<i>s1, s2, n</i>)	Copy up to n characters s2->s1
	iresult=strcspn (<i>s1, s2</i>)	Count of initial chars in s1 not in s2
	iresult=strspn (<i>s1, s2</i>)	Count of initial chars in s1 also in s2
	iresult=strlen (<i>s1</i>)	Number of characters in s1
	ptr=strlwr (<i>s1</i>)	Convert string to lower case
	ptr=strpbrk (<i>s1, s2</i>)	Search s1 for first char also in s2
	ptr= strstr (<i>s1, s2</i>)	Search for s2 in s1
	ptr=strncat(<i>s1,s2</i>)	Concatenates up to n bytes of s2 onto s1
	iresult= strcoll(<i>s1,s2</i>)	Compares s1 to s2, both interpreted as appropriate to the current locale.
	res= strxfrm(<i>s1,s2,n</i>)	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)
	iresult=memcmp(<i>m1,m2,n</i>)	Compare m1 to m2 (n bytes)
	ptr=memchr(<i>m1,c,n</i>)	Find c in first n characters of m1 and return &m1[i]
	ptr= strerror(errnum)	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
 iresult is an 8 bit int
 result is -1 (less than), 0 (equal) or 1 (greater than)
 res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples: 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 [ex_str.c](#)

Files:

Also See: strcpy(), [strtok\(\)](#)

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
	cresult=strcmp (s1, s2)	Compare s1 to s2
	iresult=strncmp (s1, s2, n)	Compare s1 to s2 (n bytes)
	iresult=stricmp (s1, s2)	Compare and ignore case
	ptr=strncpy (s1, s2, n)	Copy up to n characters s2->s1
	iresult=strcspn (s1, s2)	Count of initial chars in s1 not in s2
	iresult=strspn (s1, s2)	Count of initial chars in s1 also in s2
	iresult=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)	Concatenates up to n bytes of s2 onto s1
	iresult=strcoll(s1,s2)	Compares s1 to s2, both interpreted as appropriate to the current locale.

<code>res=strxfrm(<i>s1,s2,n</i>)</code>	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)
<code>iresult=memcmp(<i>m1,m2,n</i>)</code>	Compare m1 to m2 (n bytes)
<code>ptr=memchr(<i>m1,c,n</i>)</code>	Find c in first n characters of m1 and return &m1[i]
<code>ptr=strerror(<i>errnum</i>)</code>	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
 iresult is an 8 bit int
 result is -1 (less than), 0 (equal) or 1 (greater than)
 res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples: `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 [ex_str.c](#)

Files:

Also See: [strcpy\(\)](#), [strtok\(\)](#)

STANDARD STRING FUNCTIONS() memchr() memcmp() strcat() strchr() strcmp() strcoll() strcspn() strerror() stricmp() strlen() strlwr() strncat() strncmp() strncpy() strpbrk() strrchr() strspn() strstr() strxfrm()

Syntax: <code>ptr=strcat (<i>s1, s2</i>)</code>	Concatenate s2 onto s1
<code>ptr=strchr (<i>s1, c</i>)</code>	Find c in s1 and return &s1[i]
<code>ptr=strrchr (<i>s1, c</i>)</code>	Same but search in reverse
<code>cresult=strcmp (<i>s1, s2</i>)</code>	Compare s1 to s2
<code>iresult=strncmp (<i>s1, s2, n</i>)</code>	Compare s1 to s2 (n bytes)
<code>iresult=stricmp (<i>s1, s2</i>)</code>	Compare and ignore case

<code>ptr=strncpy (<i>s1, s2, n</i>)</code>	Copy up to n characters s2->s1
<code>iresult=strcspn (<i>s1, s2</i>)</code>	Count of initial chars in s1 not in s2
<code>iresult=strspn (<i>s1, s2</i>)</code>	Count of initial chars in s1 also in s2
<code>iresult=strlen (<i>s1</i>)</code>	Number of characters in s1
<code>ptr=striwr (<i>s1</i>)</code>	Convert string to lower case
<code>ptr=strpbrk (<i>s1, s2</i>)</code>	Search s1 for first char also in s2
<code>ptr=strstr (<i>s1, s2</i>)</code>	Search for s2 in s1
<code>ptr=strncat(<i>s1,s2</i>)</code>	Concatenates up to n bytes of s2 onto s1
<code>iresult=strcoll(<i>s1,s2</i>)</code>	Compares s1 to s2, both interpreted as appropriate to the current locale.
<code>res=strxfrm(<i>s1,s2,n</i>)</code>	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)
<code>iresult=memcmp(<i>m1,m2,n</i>)</code>	Compare m1 to m2 (n bytes)
<code>ptr=memchr(<i>m1,c,n</i>)</code>	Find c in first n characters of m1 and return &m1[i]
<code>ptr= strerror(errnum)</code>	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: `ptr` is a copy of the `s1` pointer
 `iresult` is an 8 bit int
 `result` is -1 (less than), 0 (equal) or 1 (greater than)
 `res` is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: `#include <string.h>`

Examples: `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 [ex_str.c](#)

Files:

Also See: `strcpy()`, [strtok\(\)](#)

STANDARD STRING FUNCTIONS()

memchr() **memcmp()** **strcat()**
strchr() **strcmp()** **strcoll()** **strcspn()** **strerror()** **stricmp()** **strlen()**
strlwr() **strncat()** **strncmp()** **strncpy()** **strupbrk()** **strrchr()** **strspn()**
strstr() **strxfrm()**

Syntax:	<code>ptr=strcat (s1, s2)</code> <code>ptr=strchr (s1, c)</code> <code>ptr= strrchr (s1, c)</code> <code>cresult=strcmp (s1, s2)</code> <code>iresult=strncmp (s1, s2, n)</code> <code>iresult=strcmp (s1, s2)</code> <code>ptr=strncpy (s1, s2, n)</code> <code>iresult=strcspn (s1, s2)</code> <code>iresult=strspn (s1, s2)</code> <code>iresult= strlen (s1)</code> <code>ptr=strlwr (s1)</code> <code>ptr=strupbrk (s1, s2)</code> <code>ptr= strstr (s1, s2)</code> <code>ptr= strncat(s1,s2)</code> <code>iresult= strcoll(s1,s2)</code> <code>res=strxfrm(s1,s2,n)</code> <code>iresult=memcmp(m1,m2,n)</code> <code>ptr=memchr(m1,c,n)</code> <code>ptr= strerror(errnum)</code>	Concatenate s2 onto s1 Find c in s1 and return &s1[i] Same but search in reverse Compare s1 to s2 Compare s1 to s2 (n bytes) Compare and ignore case Copy up to n characters s2->s1 Count of initial chars in s1 not in s2 Count of initial chars in s1 also in s2 Number of characters in s1 Convert string to lower case Search s1 for first char also in s2 Search for s2 in s1 Concatenates up to n bytes of s2 onto s1 Compares s1 to s2, both interpreted as appropriate to the current locale. 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) Compare m1 to m2 (n bytes) Find c in first n characters of m1 and return &m1[i] Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.
---------	---	--

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns:

- ptr is a copy of the s1 pointer
- iresult is an 8 bit int
- result is -1 (less than), 0 (equal) or 1 (greater than)
- res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: `#include <string.h>`

Examples:

```
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 [ex_str.c](#)

Files:

Also See: [strcpy\(\)](#), [strtok\(\)](#)

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
	cresult=strcmp (s1, s2)	Compare s1 to s2
	iresult=strncmp (s1, s2, n)	Compare s1 to s2 (n bytes)
	iresult=stricmp (s1, s2)	Compare and ignore case
	ptr=strncpy (s1, s2, n)	Copy up to n characters s2->s1
	iresult=strcspn (s1, s2)	Count of initial chars in s1 not in s2
	iresult=strspn (s1, s2)	Count of initial chars in s1 also in s2
	iresult=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=strcmp (s1, s2)	Search for s2 in s1
	ptr=strncat(s1,s2)	Concatenates up to n bytes of s2 onto s1
	iresult=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)
	iresult=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[i]
	ptr=strerror(errnum)	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
 iresult is an 8 bit int
 result is -1 (less than), 0 (equal) or 1 (greater than)
 res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples:

```
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 [ex_str.c](#)

Files:

Also See: [strcpy\(\)](#), [strtok\(\)](#)

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 (<i>s1, s2</i>)	Concatenate s2 onto s1
	ptr=strchr (<i>s1, c</i>)	Find c in s1 and return &s1[i]
	ptr= strrchr (<i>s1, c</i>)	Same but search in reverse
	cresult=strcmp (<i>s1, s2</i>)	Compare s1 to s2
	iresult=strncmp (<i>s1, s2, n</i>)	Compare s1 to s2 (n bytes)
	iresult=stricmp (<i>s1, s2</i>)	Compare and ignore case
	ptr=strncpy (<i>s1, s2, n</i>)	Copy up to n characters s2->s1
	iresult=strcspn (<i>s1, s2</i>)	Count of initial chars in s1 not in s2
	iresult=strspn (<i>s1, s2</i>)	Count of initial chars in s1 also in s2
	iresult=strlen (<i>s1</i>)	Number of characters in s1
	ptr=strlwr (<i>s1</i>)	Convert string to lower case
	ptr=strpbrk (<i>s1, s2</i>)	Search s1 for first char also in s2
	ptr= strstr (<i>s1, s2</i>)	Search for s2 in s1
	ptr=strncat(<i>s1,s2</i>)	Concatenates up to n bytes of s2 onto s1
	iresult= strcoll(<i>s1,s2</i>)	Compares s1 to s2, both interpreted as appropriate to the current locale.
	res=strxfrm(<i>s1,s2,n</i>)	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)
	iresult=memcmp(<i>m1,m2,n</i>)	Compare m1 to m2 (n bytes)
	ptr=memchr(<i>m1,c,n</i>)	Find c in first n characters of m1 and return &m1[i]
	ptr=strerror(errno)	Maps the error number in errno to an error message string. The parameters 'errno' 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
 iresult is an 8 bit int
 result is -1 (less than), 0 (equal) or 1 (greater than)
 res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples: 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 [ex_str.c](#)

Files:

Also See: strcpy(), [strtok\(\)](#)

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
	cresult=strcmp (s1, s2)	Compare s1 to s2
	iresult=strncmp (s1, s2, n)	Compare s1 to s2 (n bytes)
	iresult=stricmp (s1, s2)	Compare and ignore case
	ptr=strncpy (s1, s2, n)	Copy up to n characters s2->s1
	iresult=strcspn (s1, s2)	Count of initial chars in s1 not in s2
	iresult=strspn (s1, s2)	Count of initial chars in s1 also in s2
	iresult=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)	Concatenates up to n bytes of s2 onto s1
	iresult=strcoll(s1,s2)	Compares s1 to s2, both interpreted as appropriate to the current locale.

<code>res=strxfrm(s1,s2,n)</code>	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)
<code>iresult=memcmp(m1,m2,n)</code>	Compare m1 to m2 (n bytes)
<code>ptr=memchr(m1,c,n)</code>	Find c in first n characters of m1 and return &m1[i]
<code>ptr=strerror(errnum)</code>	Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
 iresult is an 8 bit int
 result is -1 (less than), 0 (equal) or 1 (greater than)
 res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples: `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 [ex_str.c](#)

Files:

Also See: [strcpy\(\)](#), [strtok\(\)](#)

strtod() strtod() strtod48()

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

Parameters: **nptr** and **endptr** are strings

Returns: strtod returns a double precision floating point number.
 strtodf returns a single precision floating point number.
 strtod48 returns a 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 [ex_str.c](#)

Files:

Also See: [strxxxx\(\)](#), [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: signed long result;
char str[9]="123hello";
char *ptr;
result=strtol(str,&ptr,10);
//result is 123 and ptr is "hello"

Example None

Files:

Also See: [strtod\(\)](#), [strtoul\(\)](#)

strtoul()

Syntax: result= strtoul(*nptr,endptr, base*)

Parameters: ***nptr*** and ***endptr*** are strings pointers and ***base*** is an integer 2-36.

Returns: result is an unsigned long int.
returns the converted value in result , if any. If no conversion could be performed, zero is returned.

Function: The strtoul function converts the initial portion of the string pointed to by nptr to a long int representation in some radix determined by the value of base. The part of the string after conversion is stored in the object pointed to endptr, provided that endptr is not a null pointer. If nptr is empty or does not have the expected form, no conversion is performed and the value of nptr is stored in the object pointed to by endptr, provided endptr is not a null pointer.

Availability: All devices.

Requires: STDLIB.H must be included

Examples:

```
long result;
char str[9] = "123hello";
char *ptr;
result= strtoul(str,&ptr,10);
//result is 123 and ptr is "hello"
```

Example None

Files:

Also See: [strtol\(\)](#), [strtod\(\)](#)

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 (<i>s1, s2</i>)	Concatenate s2 onto s1
	ptr=strchr (<i>s1, c</i>)	Find c in s1 and return &s1[i]
	ptr=strrchr (<i>s1, c</i>)	Same but search in reverse
	cresult=strcmp (<i>s1, s2</i>)	Compare s1 to s2
	iresult=strncmp (<i>s1, s2, n</i>)	Compare s1 to s2 (n bytes)
	iresult=stricmp (<i>s1, s2</i>)	Compare and ignore case
	ptr=strncpy (<i>s1, s2, n</i>)	Copy up to n characters s2->s1
	iresult=strcspn (<i>s1, s2</i>)	Count of initial chars in s1 not in s2
	iresult=strspn (<i>s1, s2</i>)	Count of initial chars in s1 also in s2
	iresult=strlen (<i>s1</i>)	Number of characters in s1
	ptr=strlwr (<i>s1</i>)	Convert string to lower case
	ptr=strpbrk (<i>s1, s2</i>)	Search s1 for first char also in s2
	ptr=strstr (<i>s1, s2</i>)	Search for s2 in s1
	ptr=strncat(<i>s1,s2</i>)	Concatenates up to n bytes of s2 onto s1
	iresult=strcoll(<i>s1,s2</i>)	Compares s1 to s2, both interpreted as appropriate to the current locale.
	res=strxfrm(<i>s1,s2,n</i>)	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)
	iresult=memcmp(<i>m1,m2,n</i>)	Compare m1 to m2 (n bytes)
	ptr=memchr(<i>m1,c,n</i>)	Find c in first n characters of m1 and return &m1[i]

`ptr= strerror(errnum)` Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string.

Parameters: **s1** and **s2** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

n is a count of the maximum number of character to operate on.

c is a 8 bit character

m1 and **m2** are pointers to memory.

Returns: ptr is a copy of the s1 pointer
iresult is an 8 bit int
result is -1 (less than), 0 (equal) or 1 (greater than)
res is an integer.

Function: Functions are identified above.

Availability: All devices.

Requires: #include <string.h>

Examples: `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:

Also See: [strcpy\(\)](#), [strtok\(\)](#)

swap()

Syntax: `swap (Ivalue)`
`result = swap(Ivalue)`

Parameters: **Ivalue** 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:

```
x=0x45;
swap(x);
//x now is 0x54

int x = 0x42;
int result;
result = swap(x);
// result is 0x24;
```

Example Files: None

Also See: [rotate_right\(\)](#), [rotate_left\(\)](#)

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 [ex_tank.c](#)

Files:

Also See: [log\(\)](#), [log10\(\)](#), [exp\(\)](#), [pow\(\)](#), [sqrt\(\)](#)

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 [ex_tank.c](#)

Files:

Also See: [log\(\)](#), [log10\(\)](#), [exp\(\)](#), [pow\(\)](#), [sqrt\(\)](#)

tolower() toupper()

Syntax: result = tolower (**cvalue**)
 result = toupper (**cvalue**)

Parameters: **cvalue** is a character

Returns: An 8 bit character

Function: These functions change the case of letters in the alphabet.

TOLOWER(X) will return 'a'..'z' for X in 'A'..'Z' and all other characters are unchanged. TOUPPER(X) will return 'A'..'Z' for X in 'a'..'z' and all other characters are unchanged.

Availability: All devices.

Requires: Nothing

Examples:

```
switch( toupper(getc()) ) {  
    case 'R' : read_cmd(); break;  
    case 'W' : write_cmd(); break;  
    case 'Q' : done=TRUE; break;  
}
```

Example [ex_str.c](#)

Files:

Also See: None

touchpad_getc()

Syntax: input = TOUCHPAD_GETC();

Parameters: None

Returns: char (returns corresponding ASCII number if "input" declared as int)

Function: Actively waits for firmware to signal that a pre-declared Capacitive Sensing Module (CSM) or charge time measurement unit (CTMU) pin is active, then stores the pre-declared character value of that pin in "input".

Note: Until a CSM or CTMU pin is read by firmware as active, this instruction will cause the microcontroller to stall.

Availability: All PIC's with a CSM or CTMU Module

Requires: #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);

    c = TOUCHPAD_GETC();
    //will wait until one of declared pins is detected
    //if PIN_B0 is pressed, c will get value 'A'
}

```

Example None

Files:

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 None

Files:

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_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: **#USE_RS232(UART1,BAUD=9600,TRANSMIT_BUFFER=50)**

```
void main(void) {
    char c;

    if(!tx_buffer_full())
        putc(c);
}
```

Example Files: **None**

Also See: [USE_RS232\(\)](#), [RCV_BUFFER_FULL\(\)](#), [TX_BUFFER_FULL\(\)](#),
[RCV_BUFFER_BYTES\(\)](#), [GETC\(\)](#), [PUTC\(\)](#), [PRINTF\(\)](#), [SETUP_UART\(\)](#),
[PUTC_SEND\(\)](#)

va_arg()

Syntax: **va_arg(argptr, type)**

Parameters: **argptr** is a special argument pointer of type **va_list**

type – This is data type like int or char.

Returns: **The first call to va_arg after va_start return the value of the parameters after that specified by the last parameter. Successive invocations return the values of the remaining arguments in succession.**

Function: **The function will return the next argument every time it is called.**

Availability: All devices.

Requires: **#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**

Files:

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 None

Files:

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 (dataptr, count)`

Parameters: **dataptr**: pointer to one or more bytes
count: a 8 bit integer

Returns: undefined

Function: Erases all fuses and writes count bytes from the dataptr to the configuration memory.

Availability: All PIC18 flash devices

Requires: Nothing

Examples: `int data[6];
write_configuration_memory(data, 6)`

Example Files: None

Also See: [write_program_memory\(\)](#), Configuration Memory Overview

write_eeprom()

Syntax: `write_eeprom (address, value)`
`write_eeprom (address , pointer , N)`

Parameters: **address** is the 0 based starting location of the EEPROM write
N specifies the number of EEPROM bytes to write
value is a constant or variable to write to EEPROM
pointer is a pointer to location to data to be written to EEPROM

Returns: undefined

Function: This function will write the specified value to the given address of EEPROM. If pointers are used than the function will write n bytes of data from the pointer to EEPROM starting at the value of address.
In order to allow interrupts to occur while using the write operation, use the

#DEVICE option WRITE_EEPROM = NOINT. This will allow interrupts to occur while the write_eeprom() operations is polling the done bit to check if the write operations has completed. Can be used as long as no EEPROM operations are performed during an ISR.

Availability: This function is only available on devices with supporting hardware on chip.

Requires: Nothing

Examples:

```
#define LAST_VOLUME 10 // Location in EEPROM  
  
volume++;  
write_eeprom(LAST_VOLUME, volume);
```

Example Files: None

Also See: [read_eeprom\(\)](#), [write_program_eeprom\(\)](#), [read_program_eeprom\(\)](#), [data Eeprom Overview](#)

write_extended_ram()

Syntax: write_extended_ram (**page,address,data,count**);

Parameters: **page** – the page in extended RAM to write to
address – the address on the selected page to start writing to
data – pointer to the data to be written
count – the number of bytes to write (0-32768)

Returns: undefined

Function: To write data to the extended RAM of the PIC.

Availability: On devices with more then 30K of RAM.

Requires: Nothing

Examples:

```
unsigned int8 data[8] =  
{0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};  
  
write_extended_ram(1,0x0000,data,8);
```

Example Files: None

Also See: [read_extended_ram\(\)](#), [Extended RAM Overview](#)

write_program_memory()

Syntax: write_program_memory(**address, dataptr, count**);

Parameters: **address** is 32 bits .

dataptr is a pointer to one or more bytes

count is a 16 bit integer on PIC16 and 16-bit for PIC18

Returns: undefined

Function: Writes count bytes to program memory from dataptr to address. This function is most effective when count is a multiple of FLASH_WRITE_SIZE, but count needs to be a multiple of four. Whenever this function is about to write to a location that is a multiple of FLASH_ERASE_SIZE then an erase is performed on the whole block. Due to the 24 bit instruction length on PCD parts, every fourth byte of data is ignored. Fill the ignored bytes with 0x00.

See Program EEPROM Overview for more information on program memory access

Availability: Only devices that allow writes to program memory.

Requires: Nothing

Examples:

```
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

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)
Iconv	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 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 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 functions:	Multibyte characters not supported
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.

ERROR MESSAGES

Compiler Error Messages

ENDIF with no corresponding #IF

Compiler found a #ENDIF directive without a corresponding #IF.

#ERROR

A #DEVICE required before this line

The compiler requires a #device before it encounters any statement or compiler directive that may cause it to generate code. In general #defines may appear before a #device but not much more.

ADDRESSMOD function definition is incorrect

ADDRESSMOD range is invalid

A numeric expression must appear here

Some C expression (like 123, A or B+C) must appear at this spot in the code. Some expression that will evaluate to a value.

Arrays of bits are not permitted

Arrays may not be of SHORT INT. Arrays of Records are permitted but the record size is always rounded up to the next byte boundary.

Assignment invalid: value is READ ONLY

Attempt to create a pointer to a constant

Constant tables are implemented as functions. Pointers cannot be created to functions. For example CHAR CONST MSG[9]={"HI THERE"}; is permitted, however you cannot use &MSG.

You can only reference MSG with subscripts such as MSG[i] and in some function calls such as Printf and STRCPY.

Attributes used may only be applied to a function (INLINE or SEPARATE)

An attempt was made to apply #INLINE or #SEPARATE to something other than a function.

Bad ASM syntax

Bad expression syntax

This is a generic error message. It covers all incorrect syntax.

Baud rate out of range

The compiler could not create code for the specified baud rate. If the internal UART is being used the combination of the clock and the UART capabilities could not get a baud rate within 3% of the requested value. If the built in UART is not being used then the clock will not permit the indicated baud rate. For fast baud rates, a faster clock will be required.

BIT variable not permitted here

Addresses cannot be created to bits. For example &X is not permitted if X is a SHORT INT.

Branch out of range

Cannot change device type this far into the code

The #DEVICE is not permitted after code is generated that is device specific. Move the #DEVICE to an area before code is generated.

Character constant constructed incorrectly

Generally this is due to too many characters within the single quotes. For example 'ab' is an error as is 'nr'. The backslash is permitted provided the result is a single character such as '\010' or '\n'.

Constant out of the valid range

This will usually occur in inline assembly where a constant must be within a particular range and it is not. For example BTFSC 3,9 would cause this error since the second operand must be from 0-8.

Data item too big

Define expansion is too large

A fully expanded DEFINE must be less than 255 characters. Check to be sure the DEFINE is not recursively defined.

Define syntax error

This is usually caused by a missing or misplaced (or) within a define.

Demo period has expired

Please contact CCS to purchase a licensed copy.

www.ccsinfo.com/pricing

Different levels of indirection

This is caused by a INLINE function with a reference parameter being called with a parameter that is not a variable. Usually calling with a constant causes this.

Divide by zero

An attempt was made to divide by zero at compile time using constants.

Duplicate case value

Two cases in a switch statement have the same value.

Duplicate DEFAULT statements

The DEFAULT statement within a SWITCH may only appear once in each SWITCH. This error indicates a second DEFAULT was encountered.

Duplicate function

A function has already been defined with this name. Remember that the compiler is not case sensitive unless a #CASE is used.

Duplicate Interrupt Procedure

Only one function may be attached to each interrupt level. For example the #INT_RB may only appear once in each program.

Element is not a member

A field of a record identified by the compiler is not actually in the record. Check the identifier spelling.

ELSE with no corresponding IF

Compiler found an ELSE statement without a corresponding IF. Make sure the ELSE statement always match with the previous IF statement.

End of file while within define definition

The end of the source file was encountered while still expanding a define. Check for a missing).

End of source file reached without closing comment */ symbol

The end of the source file has been reached and a comment (started with /*) is still in effect.

The */ is missing.

type are INT and CHAR.

Expect ;

Expect }

Expect CASE

Expect comma

Expect WHILE

Expecting *

Expecting :

Expecting <

Expecting =

Expecting >

Expecting a (

Expecting a , or)

Expecting a , or }

Expecting a .

Expecting a ; or ,

Expecting a ; or {

Expecting a close paren
Expecting a declaration
Expecting a structure/union
Expecting a variable
Expecting an =
Expecting a]
Expecting a {
Expecting an array
Expecting an identifier
Expecting function name
Expecting an opcode mnemonic
This must be a Microchip mnemonic such as MOVLW or BTFSC.
Expecting LVALUE such as a variable name or * expression
This error will occur when a constant is used where a variable should be. For example 4=5; will give this error.
Expecting a basic type
Examples of a basic type are INT and CHAR.

Expression must be a constant or simple variable
The indicated expression must evaluate to a constant at compile time. For example 5*3+1 is permitted but 5*x+1 where X is a INT is not permitted. If X were a DEFINE that had a constant value then it is permitted.
Expression must evaluate to a constant
The indicated expression must evaluate to a constant at compile time. For example 5*3+1 is permitted but 5*x+1 where X is a INT is not permitted. If X were a DEFINE that had a constant value then it is permitted.
Expression too complex
This expression has generated too much code for the compiler to handle for a single expression. This is very rare but if it happens, break the expression up into smaller parts.
Too many assembly lines are being generated for a single C statement. Contact CCS to increase the internal limits.

EXTERNal symbol not found

EXTERNal symbol type mis-match
Extra characters on preprocessor command line
Characters are appearing after a preprocessor directive that do not apply to that directive.
Preprocessor commands own the entire line unlike the normal C syntax. For example the following is an error:
`#PRAGMA DEVICE <PIC16C74> main() { int x; x=1; }`
File cannot be opened
Check the filename and the current path. The file could not be opened.
File cannot be opened for write
The operating system would not allow the compiler to create one of the output files. Make sure the file is not marked READ ONLY and that the compiler process has write privileges to the directory and file.
Filename must start with " or <
The correct syntax of a #include is one of the following two formats:
`#include "filename.ext"
#include <filename.ext>`
This error indicates neither a " or < was found after #include.
Filename must terminate with " or; msg:' '
The filename specified in a #include must terminate with a " if it starts with a ". It must terminate with a > if it starts with a <.
Floating-point numbers not supported for this operation

A floating-point number is not permitted in the operation near the error. For example, `++F` where F is a float is not allowed.

Function definition different from previous definition

This is a mis-match between a function prototype and a function definition. Be sure that if a `#INLINE` or `#SEPARATE` are used that they appear for both the prototype and definition. These directives are treated much like a type specifier.

Function used but not defined

The indicated function had a prototype but was never defined in the program.

Identifier is already used in this scope

An attempt was made to define a new identifier that has already been defined.

Illegal C character in input file

A bad character is in the source file. Try deleting the line and re-typing it.

Import error

Improper use of a function identifier

Function identifiers may only be used to call a function. An attempt was made to otherwise reference a function. A function identifier should have a (after it.

Incorrectly constructed label

This may be an improperly terminated expression followed by a label. For example:

`x=5+`

MPLAB :

Initialization of unions is not permitted

Structures can be initialized with an initial value but UNIONS cannot be.

Internal compiler limit reached

The program is using too much of something. An internal compiler limit was reached. Contact CCS and the limit may be able to be expanded.

Internal Error - Contact CCS

This error indicates the compiler detected an internal inconsistency. This is not an error with the source code; although, something in the source code has triggered the internal error. This problem can usually be quickly corrected by sending the source files to CCS so the problem can be re-created and corrected.

In the meantime if the error was on a particular line, look for another way to perform the same operation. The error was probably caused by the syntax of the identified statement. If the error was the last line of the code, the problem was in linking. Look at the call tree for something out of the ordinary.

Interrupt handler uses too much stack

Too many stack locations are being used by an interrupt handler.

Invalid conversion from LONG INT to INT

In this case, a LONG INT cannot be converted to an INT. You can type cast the LONG INT to perform a truncation. For example:

`I = INT(LI);`

Invalid interrupt directive

Invalid parameters to built in function

Built-in shift and rotate functions (such as SHIFT_LEFT) require an expression that evaluates to a constant to specify the number of bytes.

Invalid Pre-Processor directive

The compiler does not know the preprocessor directive. This is the identifier in one of the following two places:

`#xxxxxx`

`#PRAGMA xxxx`

Invalid ORG range

The end address must be greater than or equal to the start address. The range may not overlap another range. The range may not include locations 0-3. If only one address is specified it must match the start address of a previous #org.

Invalid overload function

Invalid type conversion
Label not permitted here
Library in USE not found
The identifier after the USE is not one of the pre-defined libraries for the compiler. Check the spelling.
Linker Error: "%s" already defined in "%s"
Linker Error: ("%s'
Linker Error: Canont allocate memory for the section "%s" in the module "%s", because it overlaps with other sections.
Linker Error: Cannot find unique match for symbol "%s"
Linker Error: Cannot open file "%s"
Linker Error: COFF file "%s" is corrupt; recompile module.
Linker Error: Not enough memory in the target to reallocate the section "%s" in the module "%s".
Linker Error: Section "%s" is found in the modules "%s" and "%s" with different section types.
Linker Error: Unknown error, contact CCS support.
Linker Error: Unresolved external symbol "%s" inside the module "%s".
Linker option no compatible with prior options.
Linker Warning: Section "%s" in module "%s" is declared as shared but there is no shared memory in the target chip. The shared flag is ignored.
Linker option not compatible with prior options
Conflicting linker options are specified. For example using both the EXCEPT= and ONLY= options in the same directive is not legal.
LVALUE required
This error will occur when a constant is used where a variable should be. For example 4=5; will give this error.
Macro identifier requires parameters
A #DEFINE identifier is being used but no parameters were specified, as required. For example:
`#define min(x,y) ((x<y)?x:y)`
When called MIN must have a (--) after it such as:
`r=min(value, 6);`
Macro is defined recursively
A C macro has been defined in such a way as to cause a recursive call to itself.
Missing #ENDIF
A #IF was found without a corresponding #ENDIF.
Missing or invalid .CRG file
The user registration file(s) are not part of the download software. In order for the software to run the files must be in the same directory as the .EXE files. These files are on the original diskette, CD ROM or e-mail in a non-compressed format. You need only copy them to the .EXE directory. There is one .REG file for each compiler (PCB.REG, PCM.REG and PCH.REG).

More info:

Must have a #USE DELAY before this #USE

Must have a #USE DELAY before a #USE RS232

The RS232 library uses the DELAY library. You must have a #USE DELAY before you can do a #USE RS232.

No errors

The program has successfully compiled and all requested output files have been created.

No MAIN() function found

All programs are required to have one function with the name main().

No overload function matches

No valid assignment made to function pointer

Not enough RAM for all variables

The program requires more RAM than is available. The symbol map shows variables allocated.

The call tree shows the RAM used by each function. Additional RAM usage can be obtained by breaking larger functions into smaller ones and splitting the RAM between them.

For example, a function A may perform a series of operations and have 20 local variables declared. Upon analysis, it may be determined that there are two main parts to the calculations and many variables are not shared between the parts. A function B may be defined with 7 local variables and a function C may be defined with 7 local variables. Function A now calls B and C and combines the results and now may only need 6 variables. The savings are accomplished because B and C are not executing at the same time and the same real memory locations will be used for their 6 variables (just not at the same time). The compiler will allocate only 13 locations for the group of functions A, B, C where 20 were required before to perform the same operation.

Number of bits is out of range

For a count of bits, such as in a structure definition, this must be 1-8. For a bit number specification, such as in the #BIT, the number must be 0-7.

Only integers are supported for this operation

Option invalid

Out of ROM, A segment or the program is too large

A function and all of the INLINE functions it calls must fit into one segment (a hardware code page). For example, on the PIC16 chip a code page is 512 instructions. If a program has only one function and that function is 600 instructions long, you will get this error even though the chip has plenty of ROM left. The function needs to be split into at least two smaller functions.

Even after this is done, this error may occur since the new function may be only called once and the linker might automatically INLINE it. This is easily determined by reviewing the call tree.

If this error is caused by too many functions being automatically INLINED by the linker, simply add a #SEPARATE before a function to force the function to be SEPARATE. Separate functions can be allocated on any page that has room. The best way to understand the cause of this error is to review the call tree.

Parameters must be located in RAM

Parameters not permitted

An identifier that is not a function or preprocessor macro can not have a ' (' after it.

Pointers to bits are not permitted

Addresses cannot be created to bits. For example, &X is not permitted if X is a SHORT INT.

Previous identifier must be a pointer

A -> may only be used after a pointer to a structure. It cannot be used on a structure itself or other kind of variable.

Printf format type is invalid

An unknown character is after the % in a printf. Check the printf reference for valid formats.

Printf format (%) invalid

A bad format combination was used. For example, %lc.

Printf variable count (%) does not match actual count

The number of % format indicators in the printf does not match the actual number of variables that follow. Remember in order to print a single %, you must use %%.

Recursion not permitted

The linker will not allow recursive function calls. A function may not call itself and it may not call any other function that will eventually re-call it.

Recursively defined structures not permitted

A structure may not contain an instance of itself.

Reference arrays are not permitted

A reference parameter may not refer to an array.

Return not allowed in void function

A return statement may not have a value if the function is void.

RTOS call only allowed inside task functions

Selected part does not have ICD debug capability

STDOUT not defined (may be missing #RS 232)

An attempt was made to use a I/O function such as printf when no default I/O stream has been established. Add a #USE RS232 to define a I/O stream.

Stream must be a constant in the valid range

I/O functions like fputc, fgetc require a stream identifier that was defined in a #USE RS232.

This identifier must appear exactly as it does when it was defined. Be sure it has not been redefined with a #define.

String too long

Structure field name required

A structure is being used in a place where a field of the structure must appear. Change to the form s.f where s is the structure name and f is a field name.

Structures and UNIONS cannot be parameters (use * or &)

A structure may not be passed by value. Pass a pointer to the structure using &.

Subscript out of range

A subscript to a RAM array must be at least 1 and not more than 128 elements. Note that large arrays might not fit in a bank. ROM arrays may not occupy more than 256 locations.

This linker function is not available in this compiler version.

Some linker functions are only available if the PCW or PCWH product is installed.

This type cannot be qualified with this qualifier

Check the qualifiers. Be sure to look on previous lines. An example of this error is:

VOID X;

Too many array subscripts

Arrays are limited to 5 dimensions.

Too many constant structures to fit into available space

Available space depends on the chip. Some chips only allow constant structures in certain places. Look at the last calling tree to evaluate space usage. Constant structures will appear as functions with a @CONST at the beginning of the name.

Too many elements in an ENUM

A max of 256 elements are allowed in an ENUM.

Too many fast interrupt handlers have been defined

Too many fast interrupt handlers have been identified

Too many nested #INCLUDEs

No more than 10 include files may be open at a time.

Too many parameters

More parameters have been given to a function than the function was defined with.

Too many subscripts

More subscripts have been given to an array than the array was defined with.

Type is not defined

The specified type is used but not defined in the program. Check the spelling.

Type specification not valid for a function

This function has a type specifier that is not meaningful to a function.

Undefined identifier

Undefined label that was used in a GOTO

There was a GOTO LABEL but LABEL was never encountered within the required scope. A GOTO cannot jump outside a function.

Unknown device type

A #DEVICE contained an unknown device. The center letters of a device are always C regardless of the actual part in use. For example, use PIC16C74 not PIC16RC74. Be sure the correct compiler is being used for the indicated device. See #DEVICE for more information.

Unknown keyword in #FUSES

Check the keyword spelling against the description under #FUSES.

Unknown linker keyword

The keyword used in a linker directive is not understood.

Unknown type

The specified type is used but not defined in the program. Check the spelling.

User aborted compilation

USE parameter invalid

One of the parameters to a USE library is not valid for the current environment.

USE parameter value is out of range

One of the values for a parameter to the USE library is not valid for the current environment.

Variable never used

Variable of this data type is never greater than this constant

COMPILER WARNING MESSAGES

Compiler Warning Messages

#error/warning

Assignment inside relational expression

Although legal it is a common error to do something like if(a=b) when it was intended to do if(a==b).

Assignment to enum is not of the correct type.

This warning indicates there may be such a typo in this line:

Assignment to enum is not of the correct type

If a variable is declared as a ENUM it is best to assign to the variables only elements of the enum. For example:

```
enum colors {RED, GREEN, BLUE} color;  
...  
color = GREEN; // OK  
color = 1; // Warning 209  
color = (colors)1; //OK
```

Code has no effect

The compiler can not discern any effect this source code could have on the generated code.

Some examples:

```
1;  
a==b;  
1,2,3;
```

Condition always FALSE

This error when it has been determined at compile time that a relational expression will never be true. For example:

```
int x;  
if( x>>9 )
```

Condition always TRUE

This error when it has been determined at compile time that a relational expression will never be false. For example:

```
#define PIN_A1 41  
...  
if( PIN_A1 ) // Intended was: if( input(PIN_A1) )
```

Function not void and does not return a value

Functions that are declared as returning a value should have a return statement with a value to be returned. Be aware that in C only functions declared VOID are not intended to return a value.

If nothing is specified as a function return value "int" is assumed.

Duplicate #define

The identifier in the #define has already been used in a previous #define. To redefine an identifier use #UNDEF first. To prevent defines that may be included from multiple source do something like:

```
#ifndef ID  
#define ID text  
#endif
```

Feature not supported

Function never called

Function not void and does not return a value.

Info:

Interrupt level changed

Interrupts disabled during call to prevent re-entrancy.

Linker Warning: "%s" already defined in object "%s"; second definition ignored.

Linker Warning: Address and size of section "%s" in module "%s" exceeds maximum range for this processor. The section will be ignored.

Linker Warning: The module "%s" doesn't have a valid chip id. The module will be considered for the target chip "%s".

Linker Warning: The target chip "%s" of the imported module "%s" doesn't match the target chip "%s" of the source.

Linker Warning: Unsupported relocation type in module "%s".

Memory not available at requested location.

Operator precedence rules may not be as intended, use() to clarify

Some combinations of operators are confusing to some programmers. This warning is issued for expressions where adding() would help to clarify the meaning. For example:

```
if( x << n + 1 )
```

would be more universally understood when expressed:

```
if( x << (n + 1) )
```

Option may be wrong

Structure passed by value

Structures are usually passed by reference to a function. This warning is generated if the structure is being passed by value. This warning is not generated if the structure is less than 5 bytes. For example:

```
void myfunct( mystruct s1 ) // Pass by value - Warning
myfunct( s2 );
void myfunct( mystruct * s1 ) // Pass by reference - OK
myfunct( &s2 );
void myfunct( mystruct & s1 ) // Pass by reference - OK
myfunct( s2 );
```

Undefined identifier

The specified identifier is being used but has never been defined. Check the spelling.

Unprotected call in a #INT_GLOBAL

The interrupt function defined as #INT_GLOBAL is intended to be assembly language or very simple C code. This error indicates the linker detected code that violated the standard memory allocation scheme. This may be caused when a C function is called from a #INT_GLOBAL interrupt handler.

Unreachable code

Code included in the program is never executed. For example:

```
if(n==5)
    goto do5;
goto exit;
if(n==20) // No way to get to this line
    return;
```

Unsigned variable is never less than zero

Unsigned variables are never less than 0. This warning indicates an attempt to check to see if an unsigned variable is negative. For example the following will not work as intended:

```
int i;
for(i=10; i>=0; i--)
```

Variable assignment never used.

Variable of this data type is never greater than this constant

A variable is being compared to a constant. The maximum value of the variable could never be larger than the constant. For example the following could never be true:

```
int x; // 8 bits, 0-255
if ( x>300)
```

Variable never used

A variable has been declared and never referenced in the code.

Variable used before assignment is made.

COMMON QUESTIONS & ANSWERS

How are type conversions handled?

The compiler provides automatic type conversions when an assignment is performed. Some information may be lost if the destination can not properly represent the source. For example: int8var = int16var; Causes the top byte of int16var to be lost.

Assigning a smaller signed expression to a larger signed variable will result in the sign being maintained. For example, a signed 8 bit int that is -1 when assigned to a 16 bit signed variable is still -1.

Signed numbers that are negative when assigned to a unsigned number will cause the 2's complement value to be assigned. For example, assigning -1 to a int8 will result in the int8 being 255. In this case the sign bit is not extended (conversion to unsigned is done before conversion to more bits). This means the -1 assigned to a 16 bit unsigned is still 255.

Likewise assigning a large unsigned number to a signed variable of the same size or smaller will result in the value being distorted. For example, assigning 255 to a signed int8 will result in -1.

The above assignment rules also apply to parameters passed to functions.

When a binary operator has operands of differing types then the lower order operand is converted (using the above rules) to the higher. The order is as follows:

- Float
- Signed 32 bit
- Unsigned 32 bit
- Signed 16 bit
- Unsigned 16 bit
- Signed 8 bit
- Unsigned 8 bit
- 1 bit

The result is then the same as the operands. Each operator in an expression is evaluated independently. For example:

i32 = i16 - (i8 + i8)

The + operator is 8 bit, the result is converted to 16 bit after the addition and the - is 16 bit, that result is converted to 32 bit and the assignment is done. Note that if i8 is 200 and i16 is 400 then the result in i32 is 256. (200 plus 200 is 144 with a 8 bit +)

Explicit conversion may be done at any point with (type) inserted before the expression to be converted. For example in the above the perhaps desired effect may be achieved by doing:

i32 = i16 - ((long)i8 + i8)

In this case the first i8 is converted to 16 bit, then the add is a 16 bit add and the second i8 is forced to 16 bit.

A common C programming error is to do something like:

i16 = i8 * 100;

When the intent was:

```
i16 = (long) i8 * 100;
```

Remember that with unsigned ints (the default for this compiler) the values are never negative. For example 2-4 is 254 (in 8 bit). This means the following is an endless loop since i is never less than 0:

```
int i;  
for( i=100; i>=0; i--)
```

How can a constant data table be placed in ROM?

The compiler has support for placing any data structure into the device ROM as a constant read-only element. Since the ROM and RAM data paths are separate , there are restrictions on how the data is accessed. For example, to place a 10 element BYTE array in ROM use:

```
BYTE CONST TABLE [10] = {9,8,7,6,5,4,3,2,1,0};
```

and to access the table use:

```
x = TABLE [i];
```

OR

```
x = TABLE [5];
```

BUT NOT

```
ptr = &TABLE [i];
```

In this case, a pointer to the table cannot be constructed.

Similar constructs using CONST may be used with any data type including structures, longs and floats.

The following are two methods provided:

1. Efficient access with "const".

2. Pointer friendly "ROM" Qualifier, for example:

```
ROM BYTE TABLE[10] = {9,8,7,6,5,4,3,2,1,0}  
and to access the table use:  
    x = TABLE[i];  
    or  
    PTR = &TABLE[i];  
    and  
    x = *PTR;  
//Be sure not to mix RAM and ROM pointers. They are not  
interchangeable.
```

How can I use two or more RS-232 ports on one PIC®?

The #USE RS232 (and I2C for that matter) is in effect for GETC, PUTC, PRINTF and KBHIT functions encountered until another #USE RS232 is found.

The #USE RS232 is not an executable line. It works much like a #DEFINE.

The following is an example program to read from one RS-232 port (A) and echo the data to both the first RS-232 port (A) and a second RS-232 port (B).

```

#define RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1)
void put_to_a( char c ) {
    put(c);
}
char get_from_a( ) {
    return(getc());
}
#define RS232(BAUD=9600, XMIT=PIN_B2, RCV=PIN_B3)
void put_to_b( char b ) {
    putc(b);
}
main() {
    char c;
    put_to_a("Online\n\r");
    put_to_b("Online\n\r");
    while(TRUE) {
        c=get_from_a();
        put_to_b(c);
        put_to_a(c);
    }
}

```

The following will do the same thing but is more readable and is the recommended method:

```

#define RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1, STREAM=COM_A)
#define RS232(BAUD=9600, XMIT=PIN_B2, RCV=PIN_B3, STREAM=COM_B)

main() {
    char c;
    fprintf(COM_A,"Online\n\r");
    fprintf(COM_B,"Online\n\r");
    while(TRUE) {
        c = fgetc(COM_A);
        fputc(c, COM_A);
        fputc(c, COM_B);
    }
}

```

How do I directly read/write to internal registers?

A hardware register may be mapped to a C variable to allow direct read and write capability to the register. The following is an example using the TIMER0 register:

```

#define timer 1 = 0x 100
timer0= 128; //set timer0 to 128
while (timer 1 != 200); // wait for timer0 to reach 200

```

Bits in registers may also be mapped as follows:

```

#define T 1 IF = 0x 84.3
.
.
.
while (!T 1 IF); //wait for timer0 interrupt

```

Registers may be indirectly addressed as shown in the following example:

```

printf ("enter address:");
a = gethex ();
printf ("\r\n value is %x\r\n", *a);

```

The compiler has a large set of built-in functions that will allow one to perform the most common tasks with C function calls. When possible, it is best to use the built-in functions rather than directly write to registers. Register locations change between chips and some register operations require a specific algorithm to be performed when a register value is changed. The compiler also takes into account known chip errata in the implementation of the built-in functions. For example, it is better to do set_tris_B(0); rather than *0x02C6 =0;

How do I do a printf to a string?

The following is an example of how to direct the output of a printf to a string. We used the \f to indicate the start of the string.

This example shows how to put a floating point number in a string.

```
main() {
    char string[20];
    float f;
    f=12.345;
    sprintf(string, "\f%6.3f", f);
}
```

How do I get getc() to timeout after a specified time?

GETC will always wait for a character to become available unless a timeout time is specified in the #use rs232().

The following is an example of how to setup the PIC to timeout when waiting for an RS232 character.

```
#include <18F4520.h>
#fuses HS,NOWDT
#use delay(clock=20MHz)
#use rs232(UART1,baud=9600,timeout=500) //timeout = 500 milliseconds, 1/2
second
void main()
{
    char c;

    while(TRUE)
    {
        c=getc(); //if getc() timeouts 0 is returned to c
                  //otherwise receive character is returned to c

        if(c) //if not zero echo character back
            putc(c);
        //user to do code
        output_toggle(PIN_A5);
    }
}
```

How do I make a pointer to a function?

The compiler does not permit pointers to functions so that the compiler can know at compile time the complete call tree. This is used to allocate memory for full RAM re-use. Functions that could not be in execution at the same time will use the same RAM locations. In addition since there is no data stack in the PIC®, function parameters are passed in a special way that requires knowledge at compile time of what function is being called. Calling a function via a pointer will prevent knowing both of these things at compile time. Users sometimes will want function pointers to create a state machine. The following is an example of how to do this without pointers:

```
enum tasks {taskA, taskB, taskC};  
run_task(tasks task_to_run) {  
    switch(task_to_run) {  
        case taskA : taskA_main(); break;  
        case taskB : taskB_main(); break;  
        case taskC : taskC_main(); break;  
    }  
}
```

How do I wait only a specified time for a button press?

The following is an example of how to wait only a specific time for a button press.

```
#define PUSH_BUTTON PIN_A4  
int1 timeout_error;  
int1 timed_get_button_press(void){  
    int16 timeout;  
  
    timeout_error=FALSE;  
    timeout=0;  
    while(input(PUSH_BUTTON) && (++timeout<50000)) // 1/2 second  
        delay_us(10);  
    if(!input(PUSH_BUTTON))  
        return(TRUE); //button pressed  
    else{  
        timeout_error=TRUE;  
        return(FALSE); //button not pressed timeout occurred  
    }  
}
```

How do I write variables to EEPROM that are not a word?

The following is an example of how to read and write a floating point number from/to EEPROM. The same concept may be used for structures, arrays or any other types.

- n is an offset into the EEPROM

```
WRITE_FLOAT_EEPROM(int16 n, float data) {  
    write_eeprom(n, data, sizeof(float));  
}  
  
float READ_FLOAT_EEPROM(int16 n) {  
    float data;  
    (int32)data = read_eeprom(n, sizeof(float));
```

```

        return(data);
}

```

How does one map a variable to an I/O port?

Two methods are as follows:

```

#define PORTB    0x02C8 //Just an example, check the
#define ALL_OUT 0          //DATA sheet for the correct
#define ALL_IN  0xff        //address for your chip
main() {
    int i;

    set_tris_b(ALL_OUT);
    PORTB = 0; // Set all pins low
    for(i=0;i<=127;++i) // Quickly count from 0 to 127
        PORTB=i; // on the I/O port pin
    set_tris_b(ALL_IN);
    i = PORTB; // i now contains the portb value.
}

```

Remember when using the #BYTE, the created variable is treated like memory. You must maintain the tri-state control registers yourself via the SET_TRIS_X function. Following is an example of placing a structure on an I/O port:

```

struct  port_b_layout
{int data : 4;
 int rw : 1;
 int cd : 1;

};

struct  port_b_layout  port_b;
#define port_b  0x02C8
struct port_b_layout const INIT_1 = {0, 1,1, };
struct port_b_layout const INIT_2 = {3, 1,1, };
struct port_b_layout const INIT_3 = {0, 0,0, };
struct port_b_layout const FOR_SEND = {0,0,0, };
                                // All outputs
struct  port_b_layout  const FOR_READ = {15,0,0, };
                                // Data is an input
main() {
    int x;
    set_tris_b((int)FOR_SEND); // The constant
                                // structure is
                                // treated like
                                // a byte and
                                // is used to
                                // set the data
                                // direction
    port_b = INIT_1;
    delay_us(25);

    port_b = INIT_2;           // These constant structures delay_us(25);
                            // are used to set all fields
    port_b = INIT_3;           // on the port with a single
                                // command

    set_tris_b((int)FOR_READ);
    port_b.rw=0;               // Here the individual

```

```

port_b.cd=1;           // fields are accessed
                      // independently.
x = port_b.data;
}

```

How does the compiler determine TRUE and FALSE on expressions?

When relational expressions are assigned to variables, the result is always 0 or 1.

For example:

```

bytevar = 5>0;      //bytevar will be 1
bytevar = 0>5;      //bytevar will be 0

```

The same is true when relational operators are used in expressions.

For example:

```
bytevar = (x>y) *4;
```

is the same as:

```

if( x>y )
    bytevar=4;
else
    bytevar=0;

```

SHORT INTs (bit variables) are treated the same as relational expressions. They evaluate to 0 or 1.

When expressions are converted to relational expressions or SHORT INTs, the result will be FALSE (or 0) when the expression is 0, otherwise the result is TRUE (or 1).

For example:

```

bytevar = 54;
bitvar = bytevar;      //bitvar will be 1 (bytevar != 0)
if(bytevar)            //will be TRUE
bytevar = 0;
bitvar = bytevar;      //bitvar will be 0

```

How does the PIC® connect to a PC?

A level converter should be used to convert the TTL (0-5V_L) levels that the PIC® operates with to the RS-232 voltages (+/- 3-12V) used by the PIC®. The following is a popular configuration using the MAX232 chip as a level converter.

How does the PIC® connect to an I2C device?

Two I/O lines are required for I2C. Both lines must have pullup registers. Often the I2C device will have a H/W selectable address. The address set must match the address in S/W. The example programs all assume the selectable address lines are grounded.

How much time do math operations take?

Unsigned 8 bit operations are quite fast and floating point is very slow. If possible consider fixed point instead of floating point. For example instead of "float cost_in_dollars;" do "long cost_in_cents;". For trig formulas consider a lookup table instead of real time calculations (see EX_SINE.C for an example). The following are some rough times on a 24-bit PIC®. Note times will vary depending on memory banks used.

80mhz dsPIC33FJ (40MIPS)

	int8 [us]	int16 [us]	int32 [us]	int48 [us]	int64 [us]	float32 [us]	float48 [us]	float 64 [us]
+	0.075	0.75	0.175	0.275	0.375	3.450	3.825	5.025
-	0.125	0.125	0.200	0.350	0.400	3.375	3.725	5.225
*	0.175	0.100	1.150	1.850	1.975	2.450	2.950	4.525
/	0.650	0.550	13.500	25.550	68.225	12.475	22.575	33.80
exp()	*	*	*	*	*	70.675	158.55	206.125
ln()	*	*	*	*	*	94.475	157.400	201.825
sin()	*	*	*	*	*	77.875	136.925	184.225

What can be done about an OUT OF RAM error?

The compiler makes every effort to optimize usage of RAM. Understanding the RAM allocation can be a help in designing the program structure. The best re-use of RAM is accomplished when local variables are used with lots of functions. RAM is re-used between functions not active at the same time. See the NOT ENOUGH RAM error message in this manual for a more detailed example.

RAM is also used for expression evaluation when the expression is complex. The more complex the expression, the more scratch RAM locations the compiler will need to allocate to that expression. The RAM allocated is reserved during the execution of the entire function but may be re-used between expressions within the function. The total RAM required for a function is the sum of the parameters, the local variables and the largest number of scratch locations required for any expression within the function. The RAM required for a function is shown in the call tree after the RAM=. The RAM stays used when the function calls another function and new RAM is allocated for the new function. However when a function RETURNS the RAM may be re-used by another function called by the parent. Sequential calls to functions each with their own local variables is very efficient use of RAM as opposed to a large function with local variables declared for the entire process at once.

Be sure to use SHORT INT (1 bit) variables whenever possible for flags and other boolean variables. The compiler can pack eight such variables into one byte location. The compiler does this automatically whenever you use SHORT INT. The code size and ROM size will be smaller.

Finally, consider an external memory device to hold data not required frequently. An external 8 pin EEPROM or SRAM can be connected to the PIC® with just 2 wires and provide a great deal of additional storage capability. The compiler package includes example drivers for these devices. The primary drawback is a slower access time to read and write the data. The SRAM will have fast read and write with memory being lost when power fails. The EEPROM will have a very long write cycle, but can retain the data when power is lost.

What is an easy way for two or more PICs® to communicate?

There are two example programs (EX_PBUSM.C and EX_PBUSR.C) that show how to use a simple one-wire interface to transfer data between PICs®. Slower data can use pin B0 and the EXT interrupt. The built-in UART may be used for high speed transfers. An RS232 driver chip may be used for long distance operations. The RS485 as well as the high speed UART require 2 pins and minor software changes. The following are some hardware configurations.

What is an easy way for two or more PICs® to communicate?

There are two example programs (EX_PBUSM.C and EX_PBUSR.C) that show how to use a simple one-wire interface to transfer data between PICs®. Slower data can use pin B0 and the EXT interrupt. The built-in UART may be used for high speed transfers. An RS232 driver chip may be used for long distance operations. The RS485 as well as the high speed UART require 2 pins and minor software changes. The following are some hardware configurations.

What is the format of floating point numbers?

The CCS PCD compiler uses the IEEE format for all the floating point number operations. The following floating point numbers are supported:

- 32 bit floating point numbers – Single Precision
- 48 bit floating point numbers – Extended Precision
- 64 bit floating point numbers – Double Precision

The format of these numbers is as follows:

32 bit floating point numbers – Single Precision

Sign	Exponent	Ex	Mantissa	Mantissa
31	30	23	22	15 0

- 23 bit Mantissa (Bit 0 – Bit 22)
- 8 bit exponent (Bit 23 – bit 30)
- 1 sign bit (Bit 31)

Example Numbers Representation
 Hex - 32 bit float

0	0000	0000
1	3F80	0000
-1	BF80	0000
10	4120	0000
100	42C8	0000
123.45	42F6	E666
123.45E20	6427	4E53
213.45E-20	21B6	2E17
	31	15 0

48 bit floating point numbers –Extended Precision

Sign	Exponent	Mantissa	Mantissa	Mantissa
47	46	39	38	32

- 1 Sign bit – (Bit 47)
- 8 bit Exponent (Bits 39 – 46)
- 39 bit Mantissa (Bit 0 – bit 39)

Example Numbers Representation Hex - 64 bit float

1	3F80	0000	0000
-1	BF80	0000	0000
10	4120	0000	0000
100	42C8	0000	0000
123.45	42F6	E666	6666
123.45E20	6427	4E52	9759
213.45E-20	21B6	2E17	64FF
	47	31	15 0

64 bit floating point numbers – Double Precision

Sign	Exponent	Mantissa	Mantissa	Mantissa
63	62	52	51	32

- 1 Sign bit – (Bit 47)
- 11 bit Exponent (Bits 52 – 62)
- 52 bit Mantissa (Bit 0 – bit 51)

Example Numbers Representation Hex - 64 bit float

0	0000	0000	0000	0000
1	3FF0	0000	0000	0000
-1	BFF0	0000	0000	0000
10	4024	0000	0000	0000
100	4059	0000	0000	0000
123.45	405E	DCCC	CCCC	CCCC
123.45E20	4484	E9CA	52EB	182A
213.45E-20	3C36	C5C2	EC9F	DBFD
	63	47	31	15 0

Why does the .LST file look out of order?

The list file is produced to show the assembly code created for the C source code. Each C source line has the corresponding assembly lines under it to show the compiler's work. The following three special cases make the .LST file look strange to the first time viewer.

Understanding how the compiler is working in these special cases will make the .LST file appear quite normal and very useful.

1. Stray code near the top of the program is sometimes under what looks like a non-executable source line.

Some of the code generated by the compiler does not correspond to any particular source line. The compiler will put this code either near the top of the program or sometimes under a #USE that caused subroutines to be generated.

2. The addresses are out of order.

The compiler will create the .LST file in the order of the C source code. The linker has re-arranged the code to properly fit the functions into the best code pages and the best half of a code page. The resulting code is not in source order. Whenever the compiler has a discontinuity in the .LST file, it will put a * line in the file. This is most often seen between functions and in places where INLINE functions are called. In the case of an INLINE function, the addresses will continue in order up where the source for the INLINE function is located.

3. The compiler has gone insane and generated the same instruction over and over.

For example:

```
.....A=0;
03F:    CLRF  15
*
46:CLRF  15
*
051:    CLRF  15
*
113:    CLRF  15
```

This effect is seen when the function is an INLINE function and is called from more than one place. In the above case, the A=0 line is in an INLINE function called in four places. Each place it is called from gets a new copy of the code. Each instance of the code is shown along with the original source line, and the result may look unusual until the addresses and the * are noticed.

Why is the RS-232 not working right?

1. The PIC® is Sending Garbage Characters.

- A. Check the clock on the target for accuracy. Crystals are usually not a problem but RC oscillators can cause trouble with RS-232. Make sure the #USE DELAY matches the actual clock frequency.
- B. Make sure the PC (or other host) has the correct baud and parity setting.
- C. Check the level conversion. When using a driver/receiver chip, such as the MAX 232, do not use INVERT when making direct connections with resistors and/or diodes. You probably need the INVERT option in the #USE RS232.
- D. Remember that PUTC(6) will send an ASCII 6 to the PC and this may not be a visible character. PUTC('A') will output a visible character A.

2. The PIC® is Receiving Garbage Characters.

- A. Check all of the above.

3. Nothing is Being Sent.

- A. Make sure that the tri-state registers are correct. The mode (standard, fast, fixed) used will be whatever the mode is when the #USE RS232 is encountered. Staying with the default STANDARD mode is safest.
- B. Use the following main() for testing:

```

main() {
    while(TRUE)
        putc('U');
}

```

Check the XMIT pin for activity with a logic probe, scope or whatever you can. If you can look at it with a scope, check the bit time (it should be 1/BAUD). Check again after the level converter.

4. Nothing is being received.

First be sure the PIC® can send data. Use the following main() for testing:

```

main() {
    printf("start");
    while(TRUE)
        putc( getc()+1 );
}

```

When connected to a PC typing A should show B echoed back.

If nothing is seen coming back (except the initial "Start"), check the RCV pin on the PIC® with a logic probe. You should see a HIGH state and when a key is pressed at the PC, a pulse to low. Trace back to find out where it is lost.

5. The PIC® is always receiving data via RS-232 even when none is being sent.

- A. Check that the INVERT option in the USE RS232 is right for your level converter. If the RCV pin is HIGH when no data is being sent, you should NOT use INVERT. If the pin is low when no data is being sent, you need to use INVERT.
- B. Check that the pin is stable at HIGH or LOW in accordance with A above when no data is being sent.
- C. When using PORT A with a device that supports the SETUP_ADC_PORTS function make sure the port is set to digital inputs. This is not the default. The same is true for devices with a comparator on PORT A.

6. Compiler reports INVALID BAUD RATE.

- A. When using a software RS232 (no built-in UART), the clock cannot be really slow when fast baud rates are used and cannot be really fast with slow baud rates. Experiment with the clock/baud rate values to find your limits.
- B. When using the built-in UART, the requested baud rate must be within 3% of a rate that can be achieved for no error to occur. Some parts have internal bugs with BRGH set to 1 and the compiler will not use this unless you specify BRGH1OK in the #USE RS232 directive.

EXAMPLE PROGRAMS

EXAMPLE PROGRAMS

A large number of example programs are included with the software. The following is a list of many of the programs and some of the key programs are re-printed on the following pages. Most programs will work with any chip by just changing the #INCLUDE line that includes the device information. All of the following programs have wiring instructions at the beginning of the code in a comment header. The SLOW.EXE program included in the program directory may be used to demonstrate the example programs. This program will use a PC COM port to communicate with the target.

Generic header files are included for the standard PIC® parts. These files are in the DEVICES directory. The pins of the chip are defined in these files in the form PIN_B2. It is recommended that for a given project, the file is copied to a project header file and the PIN_xx defines be changed to match the actual hardware. For example; LCDRW (matching the mnemonic on the schematic). Use the generic include files by placing the following in your main .C file:

```
#include <16C74.H>
```

LIST OF COMPLETE EXAMPLE PROGRAMS (in the EXAMPLES directory)

EX_1920.C

Uses a Dallas DS1920 button to read temperature

EX_AD12.C

Shows how to use an external 12 bit A/D converter

EX_ADMM.C

A/D Conversion example showing min and max analog readings

EX_ADMM10.C

Similar to ex_admm.c, but this uses 10bit A/D readings.

EX_ADMM_STATS.C

Similar to ex_admm.c, but this uses also calculates the mean and standard deviation.

EX_BOOTLOAD.C

A stand-alone application that needs to be loaded by a bootloader (see ex_bootloader.c for a bootloader).

EX_BOOTLOADER.C

A bootloader, loads an application onto the PIC (see ex_bootload.c for an application).

EX_CAN.C

Receive and transmit CAN packets.

EX_CHECKSUM.C

Determines the checksum of the program memory, verifies it agains the checksum that was written to the USER ID location of the PIC.

EX_COMP.C

Uses the analog comparator and voltage reference available on some PIC 24 s

EX_CRC.C

Calculates CRC on a message showing the fast and powerful bit operations

EX_CUST.C

Change the nature of the compiler using special preprocessor directives

EX_FIXED.C

Shows fixed point numbers

EX_DPOT.C

Controls an external digital POT

EX_DTMF.C

Generates DTMF tones

EX_ENCOD.C

Interfaces to an optical encoder to determine direction and speed

EX_EXPIO.C

Uses simple logic chips to add I/O ports to the PIC

EX_EXSIO.C

Shows how to use a multi-port external UART chip

EX_EXTEE.C

Reads and writes to an external EEPROM

EX_EXTDYNMEM.C

Uses addressmod to create a user defined storage space, where a new qualifier is created that reads/writes to an extrenal RAM device.

EX_FAT.C

An example of reading and writing to a FAT file system on an MMC/SD card.

EX_FLOAT.C

Shows how to use basic floating point

EX_FREQC.C

A 50 mhz frequency counter

EX_GLCD.C

Displays contents on a graphic LCD, includes shapes and text.

EX_GLINT.C

Shows how to define a custom global interrupt hander for fast interrupts

EX_HUMIDITY.C

How to read the humidity from a Humirel HT3223/HTF3223 Humidity module

EX_ICD.C

Shows a simple program for use with Microchips ICD debugger

EX_INPUTCAPTURE.C

Uses the PIC input capture module to measure a pulse width

EX_INTEE.C

Reads and writes to the PIC internal EEPROM

EX_LCDKB.C

Displays data to an LCD module and reads data for keypad

EX_LCDTH.C

Shows current, min and max temperature on an LCD

EX_LED.C

Drives a two digit 7 segment LED

EX_LOAD.C

Serial boot loader program

EX_LOGGER.C

A simple temperature data logger, uses the flash program memory for saving data

EX_MACRO.C

Shows how powerful advanced macros can be in C

EX_MALLOC.C

An example of dynamic memory allocation using malloc().

EX_MCR.C

An example of reading magnetic card readers.

EX_MMCS.D.C

An example of using an MMC/SD media card as an external EEPROM. To use this card with a FAT file system, see ex_fat.c

EX_MODBUS_MASTER.C

An example MODBUS application, this is a master and will talk to the ex_modbus_slave.c example.

EX_MODBUS_SLAVE.C

An example MODBUS application, this is a slave and will talk to the ex_modbus_master.c example.

EX_MOUSE.C

Shows how to implement a standard PC mouse on a PIC

EX_MXRAM.C

Shows how to use all the RAM on parts with problem memory allocation

EX_OUTPUTCOMPARE.C

Generates a precision pulse using the PIC output compare module.

EX_PATG.C

Generates 8 square waves of different frequencies

EX_PBUSM.C

Generic PIC to PIC message transfer program over one wire

EX_PBUSR.C

Implements a PIC to PIC shared RAM over one wire

EX_PBU.TT.C

Shows how to use the B port change interrupt to detect pushbuttons

EX_PGEN.C

Generates pulses with period and duty switch selectable

EX_PLL.C

Interfaces to an external frequency synthesizer to tune a radio

EX_PSP.C

Uses the PIC PSP to implement a printer parallel to serial converter

EX_PULSE.C

Measures a pulse width using timer0

EX_PWM.C

Uses the PIC output compare module to generate a PWM pulse stream.

EX_QSORT.C

An example of using the stdlib function qsort() to sort data. Pointers to functions is used by qsort() so the user can specify their sort algorithm.

EX.REACT.C

Times the reaction time of a relay closing using the input capture module.

EX_RFID.C

An example of how to read the ID from a 125kHz RFID transponder tag.

EX_RMSDB.C

Calculates the RMS voltage and dB level of an AC signal

EX_RS485.C

An application that shows a multi-node communication protocol commonly found on RS-485 busses.

EX_RTC.C

Sets and reads an external Real Time Clock using RS232

EX_RTCLK.C

Sets and reads an external Real Time Clock using an LCD and keypad

EX_RTCTIMER.C

How to use the PIC's hardware timer as a real time clock.

EX_RTOS_DEMO_X.C

9 examples are provided that show how to use CCS's built-in RTOS (Real Time Operating System).

EX_SINE.C

Generates a sine wave using a D/A converter

EX_SISR.C

Shows how to do RS232 serial interrupts

EX_STISR.C

Shows how to do RS232 transmit buffering with interrupts

EX_SLAVE.C

Simulates an I2C serial EEPROM showing the PIC slave mode

EX_SPEED.C

Calculates the speed of an external object like a model car

EX_SPI.C

Communicates with a serial EEPROM using the H/W SPI module

EX_SPI_SLAVE.C

How to use the PIC's MSSP peripheral as a SPI slave. This example will talk to the ex_spi.c example.

EX_SQW.C

Simple Square wave generator

EX_SRAM.C

Reads and writes to an external serial RAM

EX_STEP.C

Drives a stepper motor via RS232 commands and an analog input

EX_STR.C

Shows how to use basic C string handling functions

EX_STWT.C

A stop Watch program that shows how to user a timer interrupt

EX_SYNC_MASTER.C

EX_SYNC_SLAVE.C

An example of using the USART of the PIC in synchronous mode. The master and slave examples talk to each other.

EX_TANK.C

Uses trig functions to calculate the liquid in a odd shaped tank

EX_TEMP.C

Displays (via RS232) the temperature from a digital sensor

EX_TGETC.C

Demonstrates how to timeout of waiting for RS232 data

EX_TONES.C

Shows how to generate tones by playing "Happy Birthday"

EX_TOUCH.C

Reads the serial number from a Dallas touch device

EX_USB_HID.C

Implements a USB HID device on the PIC16C765 or an external USB chip

EX_USB_SCOPE.C

Implements a USB bulk mode transfer for a simple oscilloscope on an ext ernal USB chip

EX_USB_KBMOUSE.C

EX_USB_KBMOUSE2.C

Examples of how to implement 2 USB HID devices on the same device, by combining a mouse and keyboard.

EX_USB_SERIAL.C

EX_USB_SERIAL2.C

Examples of using the CDC USB class to create a virtual COM port for backwards compatibility with legacy software.

EX_VOICE.C

Self learning text to voice program

EX_WAKUP.C

Shows how to put a chip into sleep mode and wake it up

EX_WDTDS.C

Shows how to use the dsPIC30/dsPIC33/PIC24 watchdog timer

EX_X10.C

Communicates with a TW523 unit to read and send power line X10 codes

EX_EXTA.C

The XTEA encryption cipher is used to create an encrypted link between two PICs.

LIST OF INCLUDE FILES (in the DRIVERS directory)

2401.C

Serial EEPROM functions

2402.C

Serial EEPROM functions

2404.C

Serial EEPROM functions

2408.C

Serial EEPROM functions

24128.C

Serial EEPROM functions

2416.C

Serial EEPROM functions

24256.C

Serial EEPROM functions

2432.C

Serial EEPROM functions

2465.C

Serial EEPROM functions

25160.C

Serial EEPROM functions

25320.C

Serial EEPROM functions

25640.C

Serial EEPROM functions

25C080.C

Serial EEPROM functions

68HC68R1

C Serial RAM functions

68HC68R2.C

Serial RAM functions

74165.C

Expanded input functions

74595.C

Expanded output functions

9346.C

Serial EEPROM functions

9356.C

Serial EEPROM functions

9356SPI.C

Serial EEPROM functions (uses H/W SPI)

9366.C

Serial EEPROM functions

AD7705.C

A/D Converter functions

AD7715.C

A/D Converter functions

AD8400.C

Digital POT functions

ADS8320.C

A/D Converter functions

ASSERT.H

Standard C error reporting

AT25256.C

Serial EEPROM functions

AT29C1024.C

Flash drivers for an external memory chip

CRC.C

CRC calculation functions

CE51X.C

Functions to access the 12CE51x EEPROM

CE62X.C

Functions to access the 12CE62x EEPROM

CE67X.C

Functions to access the 12CE67x EEPROM

CTYPE.H

Definitions for various character handling functions

DS1302.C

Real time clock functions

DS1621.C

Temperature functions

DS1621M.C

Temperature functions for multiple DS1621 devices on the same bus

DS1631.C

Temperature functions

DS1624.C

Temperature functions

DS1868.C

Digital POT functions

ERRNO.H

Standard C error handling for math errors

FLOAT.H

Standard C float constants

FLOATEE.C

Functions to read/write floats to an EEPROM

INPUT.C

Functions to read strings and numbers via RS232

ISD4003.C

Functions for the ISD4003 voice record/playback chip

KBD.C

Functions to read a keypad

LCD.C

LCD module functions

LIMITS.H

Standard C definitions for numeric limits

LMX2326.C

PLL functions

LOADER.C

A simple RS232 program loader

LOCALE.H

Standard C functions for local language support

LTC1298.C

12 Bit A/D converter functions

MATH.H

Various standard trig functions

MAX517.C

D/A converter functions

MCP3208.C

A/D converter functions

NJU6355.C

Real time clock functions

PCF8570.C

Serial RAM functions

SC28L19X.C

Driver for the Phillips external UART (4 or 8 port)

SETJMP.H

Standard C functions for doing jumps outside functions

STDDEF.H

Standard C definitions

STDIO.H

Not much here - Provided for standard C compatibility

STDLIB.H

String to number functions

STDLIBM.H

Standard C memory management functions

STRING.H

Various standard string functions

TONES.C

Functions to generate tones

TOUCH.C

Functions to read/write to Dallas touch devices

USB.H

Standard USB request and token handler code

USBN960X.C

Functions to interface to Nationals USBN960x USB chips

USB.C

USB token and request handler code, Also includes usb_desc.h and usb.h

X10.C

Functions to read/write X10 codes

SOFTWARE LICENSE AGREEMENT

SOFTWARE LICENSE AGREEMENT

By opening the software diskette package, you agree to abide by the following provisions. If you choose not to agree with these provisions promptly return the unopened package for a refund.

1. License- Custom Computer Services ("CCS") grants you a license to use the software program ("Licensed Materials") on a single-user computer. Use of the Licensed Materials on a network requires payment of additional fees.
2. Applications Software- Derivative programs you create using the Licensed Materials identified as Applications Software, are not subject to this agreement.
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 your purchase. CCS does not warrant that the Licensed Materials will be free from error or will meet your specific requirements.
4. Limitations- CCS makes no warranty or condition, either expressed or implied, including but not limited to any implied warranties of merchantability and fitness for a particular purpose, regarding the Licensed Materials.

Neither CCS nor any applicable licensor will be liable for an incidental or consequential damages, including but not limited to lost profits.

5. Transfers- Licensee agrees not to transfer or export the Licensed Materials to any country other than it was originally shipped to by CCS.

The Licensed Materials are copyrighted
© 1994-2010 Custom Computer Services Incorporated
All Rights Reserved Worldwide
P.O. Box 2452
Brookfield, WI 53008

INDEX

A	dma_status.....	
abs.....	E.....	175
adc.....	enable_interrupts.....	46
adc_done.....	erase_program_memory.....	177
Analog_Comparator.....	errno.h.....	47
assert.....	ERRORR.....	180
atoe.....	Example_Programs.....	184
atof.....	exp.....	184
atoi64.....	expr.....	186, 187, 188
B	ext_int_edge.....	
BANKX	Extended.RAM.....	95
BANKY	F.....	96
BIFPCD.....	fabs.....	165
bit_clear.....	File_Formats.....	189
bit_first.....	float.h.....	190
bit_last.....	floor.....	191
bit_set.....	fmod.....	192
bit_test.....	for.....	192
break.....	free.....	29
bsearch.....	frexp.....	193
C	Function_Definition.....	
Calling_the_compiler.....	FUSES.....	17
calloc	G.....	194
CAN_Bus.....	get_capturex.....	48
ceil	get_motor_pwm_count.....	195
clear_interrupt.....	get_ticks.....	196
Comment	get_timer89.....	20
Continue	get_timer9.....	29
CRC.....	get_tris_k.....	55
crc_calc.....	getc_timeout.....	199
crc_init.....	getch.....	200
D	getenv	
DAC	gets.....	56
dac_write	goto.....	202
Data EEPROM.....	goto_address.....	57
data_table.....	H.....	462
DCI	high_speed_adc_done.....	58
dci_read.....	How do I wait only a specified.....	203
dci_start.....	time for a button press?.....	204
dci_transmit_ready.....	I.....	205
dci_write.....	I/O_Part.....	206
default.....	I2C.....	27
Default_Parameters	i2c_isr_state.....	34
DEFINEDINC	i2c_poll.....	102
delay_cycles	i2c_read.....	207
delay_ms	i2c_slaveaddr.....	208
delay_us	i2c_start.....	209
DEVICE	i2c_stop.....	103
Directories.....	i2c_write.....	14
disable_interrupts.....	IF25.....	210
div	input.....	211
DMA.....	Input_Capture.....	60
dma_start.....	input_change_x.....	212

input_k	pmp_read	253.
input_state	pmp_write	252.
Internal Oscillator	port_x_pullups	65.
internal_registers	pow	464.
interrupt_active	printf	253.
interrupts	psp_overflow	66.
isamong	psp_read	257.
isxdigit	putchar	269.
itoa	puts	
K	Q	269
kbit	QEI	
L	qei_get_count	270.
label_address	qei_set_count	271.
labs	qei_status	272.
lidxp	qsort	489.
License_Agreement	R	447
locale.h	rand	273.
log	read_adc2	273.
log10	read_configuration_memory	36.
long long	read_eeprom	274.
longjmp	read_extended_ram	
M	read_high_speed_adc	276.
make16	read_program_memory	276.
make32	read_rom_memory	275.
make8	realloc	277.
malloc	RECURSIVE	47.1
math_operations	Reference_Parameters	278.
memcpy	reset_cpu	279.
memset	restart_cause	279.
modf	restart_wdt	68.
Motor Control PWM	Return	280.
mul	RQM	22.
Multiple_Project_Files	rotate_left	463.
multiple_RS-232_ports	rotate_right	
N	RS-232	281.
nargs	rtc_alarm_read	
O	rtc_alarm_write	282, 283.
offsetof	rtc_read	32.
Operator_Precedence	rtc_write	292.
output_bit	RTCC	67.
Output_Compare	rtos_await	293.
output_drive	rtos_disable	294.
output_float	rtos_enable	295.
output_high	rtos_k	284, 285, 286, 287, 288, 289, 290, 291, 292.
output_k	rtos_msg_read	296.
output_low	rtos_msg_send	297.
output_toggle	rtos_overrun	35.
Overloaded_Functions	rtos_run	
P	rtos_signal	14.
PCD_Overview	rtos_stats	297.
perror	rtos_terminate	470.
PIC_to_I2C	rtos_wait	469.
PIC_to_PC	rtos_yield	128.
PIN_SELECT	S	298.
pmp_address	Semi_Colon	299.
pmp_overflow		

set_adc_channel2	strtoul	343
set_compare_time	swap	343
set_motor_pwm_duty	T	344
set_motor_pwm_event	tanh	345
set_pullup	Technical_Support	347
set_ticks	touchpad_getc	348
set_timer89	touchpad_hit	350
set_timerx	touchpad_state	350
set_tris_k	taupper ³⁵⁶ , 357, 358, 359, 360, 361, 362, 363	175, 178, 18
set_uart_speed	Trigraph_Sequences	364
setjmp	type_conversions	365
setjmp.h	typemod	447
setup_adc_ports2	U	366
setup_adc2	ushrt_max	365
setup_capture	V	367
setup_comparator	va_arg	368
setup_compare	va_end	369
setup_crc	va_start	370
setup_dac	Variable_Parameters	371
setup_dci	W	372
setup_dma	wchar_t	373
setup_high_speed_adc	WDT_or_Watch_Dog_Timer	374
setup_high_speed_adc_pair	What_can_be_done_about_an	374
setup_low_volt_detect	QUT_OF_RAM_error?	375
setup_motor_pwm	while	376
setup_motor_unit	write_configuration_memory	345
setup_oscillator	write_eeprom	376
setup_pmp	write_extended_ram	377
setup_psp	write_program_memory	379
setup_qei		381
setup_rtc	DATE	381
setup_rtc_alarm	DEVICE	382
setup_spi	FILENAME	382
setup_timer9	LINE	383
setup_uart	PCD	389
setup_vref	TIME	390
setup_wdt	LOCATE	391
shift_left		392
shift_right	LST_file	392
sizeof	#	31
sleep	#ASM	396
spi_data_is_in2	#BANK_DMA	397
spi_read	#BIT	399
spi_write2	#BUILD	401
spi_xfer	#BYTE	401
sprintf	#CASE	403
sqrt	#DEFINE	403
srand	#EXPORT	404
standard_string_functions	404, 406, 407, 409, 410, 411ROM4, 416, 417, 418, 420, 421, 423, 424, 425, 430	
stdio.h	#HEXCOMMENT	448
stdlib.h	#ID	448
stmt	#IF	30
string	#IFDEF	465
strtod	#IGNORE_WARNINGS	427
strtok	#IMPORT	428
strtol	#INCLUDE	429

#INLINE	#UNDEF.....	117.....
#INT_DEFAULT.....	#USE_DELAY.....	120.....
#INT_UART2E.....	#USE_DYNAMIC_MEMORY.....	117.....
#LIST	#USE_FAST_IO.....	121.....
#MODULE	#USE_FIXED_IO.....	123.....
#NOLIST.....	#USE_I2C.....	125.....
#OCS.....	#USE_RS232.....	125.....
#OPT	#USE_RTOS.....	126.....
#ORG	#USE_SPI.....	126.....
#PRAGMA	#USE_STANDARD_IO.....	130.....
#RESERVE.....	#USE_TIMER.....	132.....
#SEPARATE.....	#USE_TOUCHPAD.....	134.....
#SERIALIZE	#WARNING.....	135.....
#TASK	#WORD.....	139.....
#TYPE	#ZERO_RAM.....	141.....