C Compiler Reference Manual
Version 4
January 2007

This manual documents software version 4.
Review the readme.txt file in the product directory for changes made since this version.

Copyright © 1994, 2007 Custom Computer Services, Inc.
All rights reserved worldwide. No part of this work may be reproduced or copied in any form or by any means- electronic, graphic, or mechanical, including photocopying, recording, taping, or information retrieval systems without prior permission.

Table Of Contents

Overview 1
PCB, PCM and PCH Overview 1
Installation 1
Technical Support 1
Directories 2
File Formats 2
Invoking the Command Line Compiler 3
PCW Overview 5
Program Syntax 13
Overall Structure 13
Comment 13
Trigraph Sequences 15
Multiple Files 15
Multiple Compilation Units 15
Example 15
Statements 17
Expressions 18
Operators 18
Operator Precedence 20
Reference Parameters 20
Variable Parameters 21
Default Parameters 22
Overloaded Functions 22
Data Definitions 23
Basic and Special types 23
Declarations 26
Non-RAM Data Definitions 26
Using Program Memory for Data 28
Functional Overviews 30
I2C 30
ADC 31
Analog Comparator 32
CAN Bus 32
CCP1 35
CCP2, CCP3, CCP4, CCP5, CCP6 36
Configuration Memory 36
Data EEPROM 37
External Memory 38
Internal LCD 39
Internal Oscillator 40
Interrupts 41
Low Voltage Detect 42
Power PWM 43
Program EEPROM 44
PSP 46

RS232 I/O 47
RTOS 49
SPI 51
Timer0 52
Timer1 53
Timer2 54
Timer3 54
Timer4 54
Timer5 55
USB 56
Voltage Reference 59
WDT or Watch Dog Timer 60
Pre-Processor Directives 61
#ASM, #ENDASM 63
#ENDASM 65
#BIT 66
#BUILD 67
#BYTE 68
#CASE 69
DATE 69
#DEFINE 70
#DEVICE 71
DEVICE 73
#ELIF 73
#ELSE 74
#ENDIF 74
#ERROR 74
#EXPORT (options) 75
__FILE__ 76
__FILENAME__ 77
#FILL_ROM 77
#FUSES 78
#HEXCOMMENT 79
#ID 79
#ID CHECKSUM 80
#ID "filename" 80
#ID number 16 80
#ID number, number, number, number 80
#IF exp, #ELSE, #ELIF, #ENDIF 81
#IFDEF, #IFNDEF, #ELSE, #ELIF, #ENDIF 82
#IGNORE_WARNINGS 83
#IMPORT (options) 84
#INCLUDE 85
#INLINE 85
#INT_xxxx 86
#INT_DEFAULT 89
#INT_GLOBAL 89
__LINE__ 90

#LIST 90
#LOCATE 91
#MODULE 92
#NOLIST 93
#OPT 93
#ORG 94
__PCB__ 95
__ PCM __ 96
__ PCH __ 96
#PRAGMA 97
#PRIORITY 97
#RESERVE 98
#ROM 98
#SEPARATE 99
#SERIALIZE 100
#TASK 102
__ TIME __ 103
#TYPE 103
#UNDEF 104
#USE DELAY 105
#USE FAST_IO 107
#USE FIXED_IO 107
#USE I2C 108
#USE RS232 109
#USE RTOS 112
#USE SPI 113
#USE STANDARD_IO 114
#ZERO_RAM 115
Built-in-Functions 116
ABS( ) 120
ACOS( ) 120
ASIN( ) 120
ASSERT( ) 121
ATAN( ) 121
ATAN2( ) 121
ATOF( ) 122
ATOI( ) 123
ATOI32( ) ATOL( ) 123
BIT_CLEAR( ) 124
BIT_SET( ) 125
BIT_TEST( ) 126
BSEARCH( ) 127
CALLOC( ) 128
CEIL( ) 128
CLEAR_INTERRUPT( ) 129
COS( ) 129
COSH( ) 129
DELAY_CYCLES( ) 130

DELAY_MS( ) 131
DELAY_US( ) 132
DISABLE_INTERRUPTS( ) 133
DIV( ), LDIV( ) 134
ENABLE_INTERRUPTS( ) 135
ERASE_PROGRAM_EEPROM( ) 136
EXP( ) 137
EXT_INT_EDGE( ) 138
FABS( ) 138
FGETC( ) 139
FGETS( ) 139
FLOOR( ) 139
FMOD( ) 140
FPRINTF( ) 140
FPUTC( ) 140
FPUTS( ) 141
FREE( ) 141
FREXP( ) 142
GET_TIMERx( ) 143
GET_TRISx( ) 144
GETC( ), GETCH( ), GETCHAR( ), FGETC( ) 145
GETCHAR( ) 146
GETENV( ) 146
GETS( ), FGETS( ) 148
GOTO_ADDRESS( ) 149
I2C_ISR_STATE( ) 150
I2C_POLL( ) 151
I2C_READ( ) 152
I2C_SlaveAddr( ) 153
I2C_START( ) 154
I2C_STOP( ) 155
I2C_WRITE( ) 156
INPUT( ) 157
INPUT_STATE( ) 158
INPUT_x( ) 158
INTERRUPT_ACTIVE( ) 159
ISALNUM(char), ISALPHA(char), ISDIGIT(char), ISLOWER(char), ISSPACE(char), ISUPPER(char), ISXDIGIT(char), ISCNTRL(x), ISGRAPH(x), ISPRINT(x), ISPUNCT(x) 160
ISAMOUNG( ) 161
ITOA( ) 161
JUMP_TO_ISR 162
KBHIT( ) 163
LABEL_ADDRESS( ) 164
LABS( ) 164
LCD_LOAD( ) 165
LCD_SYMBOL( ) 166
LDEXP( ) 167
LOG( ) 167

LOG10( ) 168
LONGJMP( ) 169
MAKE8( ) 169
MAKE16( ) 170
MAKE32( ) 171
MALLOC( ) 172
MEMCPY( ), MEMMOVE( ) 173
MEMSET( ) 174
MODF( ) 174
_MUL( ) 175
OFFSETOF( ), OFFSETOFBIT( ) 176
OFFSETOFBIT( ) 176
OUTPUT_A( ) 177
OUTPUT_B, OUTPUT_C, OUTPUT_D, OUTPUT_E, OUTPUT_F, OUTPUT_G, OUTPUT_H,
OUTPUT_J, OUTPUT_K 178
OUTPUT_BIT( ) 178
OUTPUT_DRIVE( ) 179
OUTPUT_FLOAT( ) 180
OUTPUT_HIGH( ) 181
OUTPUT_LOW( ) 182
OUTPUT_TOGGLE( ) 183
PERROR( ) 183
PORT_A_PULLUPS ( ) 184
PORT_B_PULLUPS( ) 184
POW( ), PWR( ) 185
PRINTF( ), FPRINTF( ) 186
PSP_OUTPUT_FULL( ), PSP_INPUT_FULL( ), PSP_OVERFLOW( ) 188
PSP_INPUT_FULL( ) 188
PSP_OVERFLOW( ) 188
PUTC( ), PUTCHAR( ), FPUTC( ) 189
PUTCHAR( ) 189
PUTS( ), FPUTS( ) 190
QSORT( ) 191
RAND( ) 192
READ_ADC( ) 193
READ_BANK( ) 194
READ_CALIBRATION( ) 195
READ_EEPROM( ) 195
READ_PROGRAM_EEPROM( ) 196
READ_PROGRAM_MEMORY( ) 196
READ_EXTERNAL_MEMORY( ) 197
REALLOC( ) 197
RESET_CPU( ) 198
RESTART_CAUSE( ) 198
RESTART_WDT( ) 199
ROTATE_LEFT( ) 200
ROTATE_RIGHT( ) 201
SET_ADC_CHANNEL( ) 202

SET_PWM1_DUTY( ) 203
SET_PWM2_DUTY, SET_PWM3_DUTY, SET_PWM4_DUTY, SET_PWM5_DUTY 204
SET_POWER_PWMX_DUTY( ) 204
SET_POWER_PWM_OVERRIDE( ) 205
SET_RTCC( ) 206
SET_TIMER0( ), SET_TIMER1( ), SET_TIMER2( ), SET_TIMER3( ), SET_TIMER4( ), SET_TIMER5( ) 206
SET_TRIS_A( ) 207
SET_TRIS_B( ), SET_TRIS_C( ), SET_TRIS_D( ), SET_TRIS_E( ), SET_TRIS_F( ), SET_TRIS_G( ), SET_TRIS_H( ), SET_TRIS_J( ), SET_TRIS_K( ) 208
SET_UART_SPEED( ) 208
SETJMP( ) 209
SETUP_ADC(mode) 209
SETUP_ADC_PORTS() 210
SETUP_CCP1( ) 211
SETUP_CCP2( ), SETUP_CCP3( ), SETUP_CCP4( ), SETUP_CCP5( ) 213
SETUP_COMPARATOR( ) 214
SETUP_COUNTERS( ) 215
SETUP_EXTERNAL_MEMORY( ) 216
SETUP_LCD( ) 217
SETUP_LOW_VOLT_DETECT( ) 218
SETUP_OSCILLATOR( ) 219
SETUP_OPAMP1( ) 220
SETUP_OPAMP2( ) 220
SETUP_POWER_PWM( ) 221
SETUP_POWER_PWM_PINS( ) 222
SETUP_PSP( ) 223
SETUP_SPI( ), SETUP_SPI2( ) 223
SETUP_TIMER_0( ) 224
SETUP_TIMER_1( ) 225
SETUP_TIMER_2( ) 226
SETUP_TIMER_3( ) 227
SETUP_TIMER_4( ) 227
SETUP_TIMER_5( ) 228
SETUP_UART( ) 229
SETUP_VREF( ) 230
SETUP_WDT( ) 231
SHIFT_LEFT( ) 232
SHIFT_RIGHT( ) 233
SIN( ), COS( ), TAN( ), ASIN( ), ACOS(), ATAN(), SINH(), COSH(), TANH(), ATAN2() 234
SINH( ) 235
SLEEP( ) 236
SLEEP_ULPWU( ) 236
SPI_DATA_IS_IN( ), SPI_DATA_IS_IN2( ) 237
SPI_READ( ), SPI_READ2( ) 238
SPI_WRITE( ), SPI_WRITE2( ) 239
SPI_XFER( ) 240
SPRINTF( ) 241

SQRT( ) 242
SRAND( ) 243
STANDARD STRING FUNCTIONS( ) 244
MEMCHR( ), MEMCMP( ), STRCAT( ), STRCHR( ), STRCMP( ), STRCOLL( ), STRCSPN( ), STRICMP( ), STRLEN( ), STRLWR( ), STRNCAT( ), STRNCMP( ), STRNCPY( ), STRPBRK( ), STRRCHR( ), STRSPN( ), STRSTR( ), STRXFRM( ) 244
STRCAT( ), STRCHR( ), STRCMP( ), STRCOLL( ) 245
STRCPY( ), STRCOPY( ) 246
STRCSPN( ), STRLEN( ), STRLWR( ), STRNCAT( ), STRNCMP( ), STRNCPY( ), STRPBRK( ), STRRCHR( ), STRSPN( ) 247
STRTOD( ) 247
STRTOK( ) 248
STRTOL( ) 249
STRTOUL( ) 250
STRXFRM( ) 251
SWAP( ) 251
TAN( ) TANH( ) 251
TOLOWER( ), TOUPPER( ) 252
WRITE_BANK( ) 253
WRITE_CONFIGURATION_MEMORY( ) 254
WRITE_EEPROM( ) 254
WRITE_EXTERNAL_MEMORY( ) 255
WRITE_PROGRAM_EEPROM( ) 256
WRITE_PROGRAM_MEMORY( ) 257
Standard C Include Files 258
errno.h 258
float.h 258
limits.h 260
locale.h 260
setjmp.h 260
stddef.h 261
stdio.h 261
stdlib.h 261
Error Messages 262
Compiler Warning Messages 274
COMMON QUESTIONS AND ANSWERS 277
How are type conversions handled? 277
How can a constant data table be placed in ROM? 278
How can I pass a variable to functions like OUTPUT_HIGH()? 279
How can I use two or more RS-232 ports on one PIC®? 280
How can the RB interrupt be used to detect a button press? 281
How do I do a printf to a string? 281
How do I directly read/write to internal registers? 282
How do I get getc() to timeout after a specified time? 282
How do I make a pointer to a function? 283
How do I put a NOP at location 0 for the ICD? 283
How do I write variables to EEPROM that are not a byte? 284
How does one map a variable to an I/O port? 284

How does the compiler determine TRUE and FALSE on expressions? 286
How does the PIC® connect to a PC? 287
How does the PIC® connect to an I2C device? 288
How much time do math operations take? 289
Instead of 800, the compiler calls 0. Why? 290
Instead of A0, the compiler is using register 20. Why? 290
What can be done about an OUT OF RAM error? 290
What is an easy way for two or more PICs® to communicate? 291
What is the format of floating point numbers? 292
Why does the .LST file look out of order? 293
Why does the compiler show less RAM than there really is? 294
Why does the compiler use the obsolete TRIS? 295
Why is the RS-232 not working right? 295
EXAMPLE PROGRAMS 297
SOFTWARE LICENSE AGREEMENT 321

OVERVIEW

PCB, PCM and PCH Overview

The PCB, PCM, and PCH are separate compilers. PCB is for 12-bit opcodes, PCM is for 14-bit opcodes, and PCH is for 16-bit opcode PIC® microcontrollers. Due to many similarities, all three compilers are covered in this reference manual. Features and limitations that apply to only specific microcontrollers are indicated within. These compilers are specifically designed to meet the unique needs of the PIC® microcontroller. This allows developers to quickly design applications software in a more readable, high-level language.

When compared to a more traditional C compiler, PCB, PCM, and PCH have some limitations. As an example of the limitations, function recursion is not allowed. This is due to the fact that the PIC® has no stack to push variables onto, and also because of the way the compilers optimize the code. The compilers can efficiently implement normal C constructs, input/output operations, and bit twiddling operations. All normal C data types are supported along with pointers to constant arrays, fixed point decimal, and arrays of bits.

Installation

PCB, PCM, and PCH Installation:
Insert the CD ROM and from Windows Start|Run type:
D:SETUP

PCW and PCWH Installation:
Insert the CD ROM, select each of the programs you wish to install and follow the on-screen instructions.

Technical Support

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

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 .PJT file
- The same directory as the source file

By default, the compiler files are put in C:\Program Files\PICC and the example programs and all Include files are in C:\Program Files\PICC\EXAMPLES.

The compiler itself is a DLL file. The DLL files are in a DLL directory by default in C:\Program Files\PICC\DLL. Old compiler versions may be kept by renaming this directory.

Compiler Version 4 and above can tolerate two compilers of different versions in the same directory. Install an older version (4.xx) and rename the devices4.dat file to devices4X.dat where X is B for PCB, M is for PCM, and H is for PCH. Install the newer compiler and do the same rename of the devices4.dat file.

File Formats

The compiler can output 8-bit hex, 16-bit hex, and binary files. Three listing formats are available: 1) Standard format resembles the Microchip tools, and may be required by other Third-Party tools. 2) Simple format is generated by compiler and is easier to read. 3) Symbolic format uses names versus addresses for registers. The debug files may be output as Microchip .COD file, Advanced Transdata .MAP file, expanded .COD file for CCS debugging or MPLAB 7.xx .COF file. All file formats and extensions may be selected via Options File Associations option in Windows IDE.

.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 project file which contains information related to the project.
.LST	This is the listing file which shows each C source line and the associated assembly code generated for that line.
.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.
.COF	This is a binary containing machine code and debugging information.
.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	This file is 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.
.OSYM	This file is generated when the compiler is set to export a relocatable object file. This file contains a list of symbols for that object.

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
+FS	Select SXC (SX)	**+DC**	Expanded .COD format debug file
+ES	Standard error file	**+EO**	Old error file format
+T	Create call tree (.TRE)	**-T**	Do not generate a tree file
+A	Create stats file (.STA)	**-A**	Do not create stats file (.STA)
+EW	Show warning messages	**-EW**	Suppress warnings (use with +EA)
+EA	Show all error messages and all warnings	**-E**	Only show first error
+Yx	Optimization level x (0-9)	**+DF**	Enables the output of a OFF debug file.

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)
+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

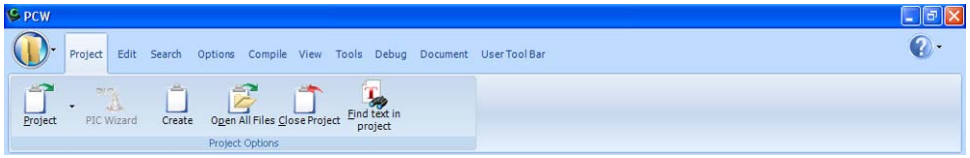
Begining in version 4.XXX of PCW, the menus and toolbars are set-up in specially organized Ribbons. Each Ribbon relates to a specific type of activity and is only shown when selected. CCS has included a "User Toolbar" Ribbon that allows the user to customize the Ribbon for individual needs.

File Menu

Click on this icon for the following items:

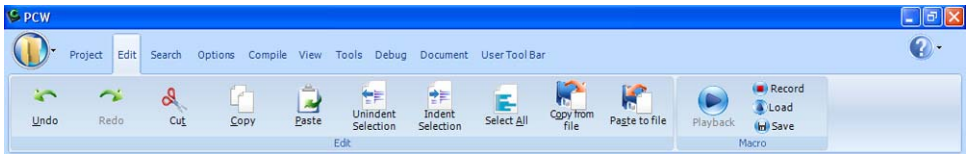
New	Creates a new File
Open	Opens a file to the editor. Includes options for Source, Project, Output, RTF, Flow Chart, Hex or Text. Ctrl+O is the shortcut.
Close	Closes the file currently open for editing. Note, that while a file is open in PCW for editing, no other program may access the file. Shift+F11 is the shortcut.
Close All	Closes all files open in the PCW.
Save	Saves the file currently selected for editing. Crtl+S is the shortcut.
Save As	Prompts for a file name to save the currently selected file.
Save All	All open files are saved.

Encrypt	Creates an encrypted include file. The standard compiler #include directive will accept files with this extension and decrypt them when read. This allows include files to be distributed without releasing the source code.
Print	Prints the currently selected file.
Recent Files	The right-side of the menu has a Recent Files list for commonly used files.
Exit	The bottom of the menu has an icon to terminate PCW.



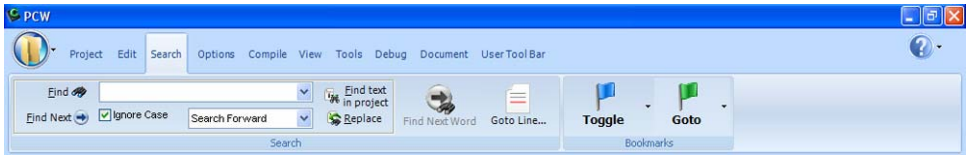
Project Menu Ribbon

Project	Open an existing project (.PJT) file as specified and the main source file is loaded.
PIC Wizard	This command is a fast way to start a new project. It will bring up a screen with fill-in-the-blanks to create a new project. When items such as RS232 I/O, i2C, timers, interrupts, A/D options, drivers and pin name are specified by the user, the Wizard will select required pins and pins that may have combined use. After all selections are made, the initial .c and .h files are created with #defines, #includes and initialization commands required for the project.
Create	Create a new project with the ability to add/remove source files, include files, global defines and specify output files.
Open All Files	Open all files in a project so that all include files become known for compilation.
Close Project	Close all files associated with project.
Find Text in Project	Ability to search all files for specific text string.



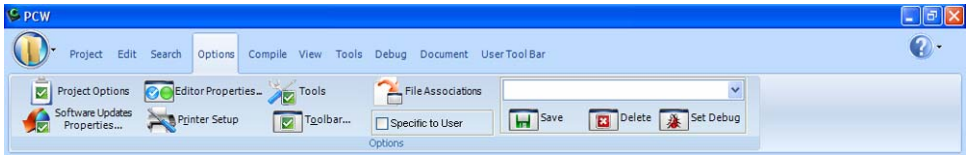
Edit Menu Ribbon

Undo	Undoes the last deletion
Redo	Re-does the last undo
Cut	Moves the selected text from the file to the clipboard.
Copy	Copies the selected text to the clipboard.
Paste	Applies the clipboard contents to the cursor location.
Unindent Selection	Selected area of code will not be indented.
Indent Selection	Selected area of code will be properly indented.
Select All	Highlighting of all text.
Copy from File	Copies the contents of a file to the cursor location.
Paste to File	Applies the selected text to a file.
Macros	Macros for recording, saving and loading keystrokes and mouse-strokes.



Search Menu Ribbon

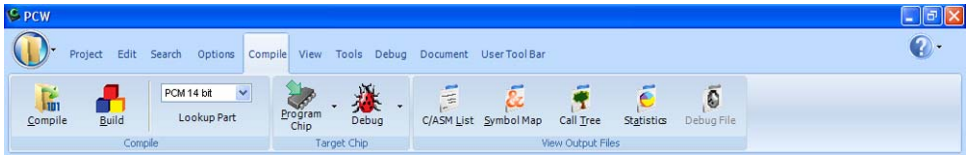
Find	Locate text in file.
Find Text in Project	Searches all files in project for specific text string.
Find Next Word at Cursor	Locates the next occurrence of the text selected in the file.
Goto Line	Cursor will move to the user specified line number.
Toggle Bookmark	Set/Remove bookmark (0-9) at the cursor location.
Goto Bookmark	Move cursor to the specified bookmark (0-9).



Options Menu Ribbon

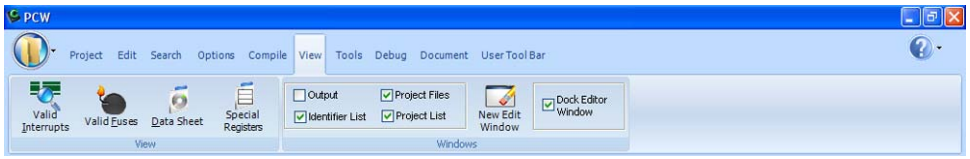
Project Options	Add/remove files, include files, global defines and output files.
Editor Properties	Allows user to define the set-up of editor properties for Windows options.

Tools	Window display of User Defined Tools and options to add and apply.
Software Updates Properties	Ability for user to select which software to update, frequency to remind user and where to archive files.
Printer Setup	Set the printer port and paper and other properties for printing.
Toolbar Setup	Customize the toolbar properties to add/remove icons and keyboard commands.
File Associations	Customize the settings for files according to software being used.



Compile Menu Ribbon

Compile	Compiles the current project in status bar using the current compiler.
Build	Compiles one or more files within a project.
Compiler	Pull-down menu to choose the compiler needed.
Lookup Part	Choose a device and the compiler needed will automatically be selected.
Program Chip	Lists the options of CCS ICD or Mach X programmers and will connect to SIOW program.
Debug	Allows for input of .hex and will output .asm for debugging.
C/ASM List	Opens listing file in read-only mode. Will show each C source line code and the associated assembly code generated.
Symbol Map	Opens the symbol file in read-only mode. Symbol map shows each register location and what program variable are saved in each location.
Call Tree	Opens the tree file in read-only mode. The call tree shows each function and what functions it calls along with the ROM and RAM usage for each.
Statistics	Opens the statistics file in read-only mode. The statistics file shows each function, the ROM and RAM usage by file, segment and name.
Debug File	Opens the debug file in read-only mode. The listing file shows each C source line code and the associated assembly code generated.



View Menu Ribbon

Valid Interrupts This displays a list of valid interrupts used with the #INT_keyword for the chip used in the current project. The interrupts for other chips can be viewed using the drop down menu.

Valid Fuses This displays a list of valid FUSE used with the #FUSES directive associated with the chip used in the current project. The fuses for other chips can be viewed using the drop down menu.

Data Sheets This tool is used to view the Manufacturer data sheets for all the Microchip parts supported by the compiler.

Part Errata This allows user to view the errata database to see what errata is associated with a part and if the compiler has compensated for the problem.

Special Registers This displays the special function registers associated with the part.

New Edit Window This will open a new edit window which can be tiled to view files side by side.

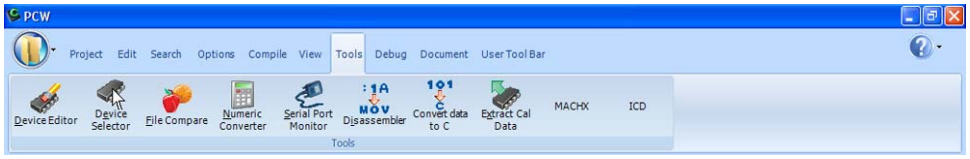
Dock Editor Window Selecting this checkbox will dock the editor window into the IDE.

Project Files When this checkbox is selected, the Project files slide out tab is displayed. This will allow quicker access to all the project source files and output files.

Project List Selecting this checkbox displays the Project slide out tab. The Project slide out tab displays all the recent project files.

Output Selecting this checkbox will enable the display of warning and error messages generated by the compiler.

Identifier List Selecting this checkbox displays the Identifier slide out tab. It allows quick access to project identifiers like functions, types, variables and defines.



Tools Menu Ribbon

Device Editor This tool is used to edit the device database used by the compiler to control compilations. The user can edit the chip memory, interrupts, fuses and other peripheral settings for all the supported devices.

Device Selector This tool uses the device database to allow for parametric selection of devices. The tool displays all eligible devices based on the selection criteria.

File Compare This utility is used to compare two files. Source or text files can be compared line by line and list files can be compared by ignoring the RAM/ROM addresses to make the comparisons more meaningful.

Numeric Converter This utility can be used to convert data between different formats. The user can simultaneously view data in various formats like binary, hex, IEEE, signed and unsigned.

Serial Port Monitor This tool is an easy way of connecting a PIC to a serial port. Data can be viewed in ASCII or hex format. An entire hex file can be transmitted to the PIC which is useful for bootloading application.

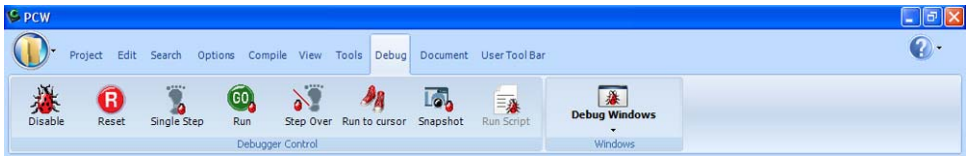
Disassembler This tool will take an input hex file and output an ASM.

Convert Data to C This utility will input data from a text file and generate code is form of a #ROM or CONST statement.

Extract Calibration This tool will input a hex file and extract the calibration data to a C include file. This feature is useful for saving calibration data stored at top of program memory from certain PIC chips.

Mach-X This will call the Mach-X.exe program and will download the hex file for the current project onto the chip.

ICD This will call the ICD.exe program and will download the hex file for the current project onto the chip.



Debug Menu Ribbon

Enable Debugger Enables the debugger. Opens the debugger window, downloads the code and on-chip debugger and resets the target into the debugger.

Reset This will reset the target into the debugger.

Single Step Executes one source code line at a time. A single line of C source code or ASM code is executed depending on whether the source code or the list file tab in the editor is active.

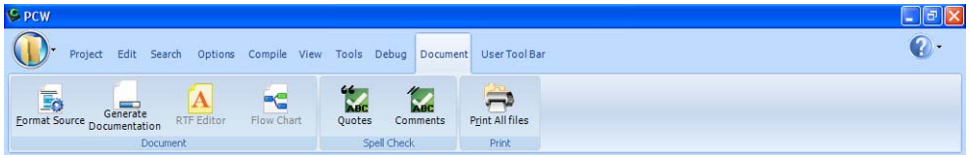
Step Over This steps over the target code. It is useful for stepping over function calls.

Run to Cursor Runs the target code to the cursor. Place the cursor at the desired location in the code and click on this button to execute the code till that address.

Snapshot This allows users to record various debugging information. Debug information like watches, ram values, data eeprom values, rom values , peripheral status can be conveniently logged. This log can be saved, printed, overwritten or appended.

Run script This tool allows the IDE's integrated debugger to execute a C-style script. The functions and variable of the program can be accesses and the debugger creates a report of the results.

Debug Windows This drop down menu allows viewing of a particular debug tab. Click on the tab name in the drop down list which you want to view and it will bring up that tab in the debugger window.



Document Tab Ribbon

Format source This utility formats the source file for indenting, color syntax highlighting, and other formatting options.

Generate Document This will call the document generator program which uses a user generated template in .RTF format to merge with comment from the source code to produce an output file in .RTF format as source code documentation.

RTF editor Open the RTF editor program which is a fully featured RTF editor to make integration of documentation into your project easier.

Flow Chart Opens a flow chart program for quick and easy charting. This tool can be used to generate simple graphics including schematics.

Quotes Performs a spell check on all the words within quotes.

Comments Performs a spell check on all the comments in your source code.

Print all files Print all the files of the current project.

Help Menu

Contents	Help File table of contents
Index	Help File index
Keyword at Cursor	Index search in Help File for the keyword at the cursor location. Press F1 to use this feature.
Debugger Help	Help File specific to debugger functionality.
Editor	Lists the Editor Keys available for use in PCW. Shft+F12 will also call this function help file page for quick review.
Data Types	Specific Help File page for basic data types.
Operators	Specific Help File page for table of operators that may be used in PCW.
Statements	Specific Help File page for table of commonly used statements.
Preprocessor Commands	Specific Help File page for listing of commonly used preprocessor commands.
Built-in Functions	Specific Help File page for listing of commonly used built-in functions provided by the compiler.
Technical Support	Technical Support wizard to directly contact Technical Support via email and the ability to attach files.
Check for Software Updates	Automatically invokes Download Manager to view local and current versions of software.
Internet	Direct links to specific CCS website pages for additional information.
About	Shows the version of compiler(s) and IDE installed.

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 the their purpose and the functions could be called from main or the subfunctions. 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
       month,   /* Current Month */
       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	Sequence	Same as
??=	#	??'	^
??(	[	??<	{
??/	\	??!	|
??)	]	??>	}
		??-	~

Multiple Files

When there are multiple files in a project they can all be included using the #include in the main file or the subfiles 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 eg: if you have main.c, x.c, x.h, y.c,y.h and z.c and z.h files in your project, you can say in:

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

Multiple Compilation Units

Traditionally the CCS C compilers used only one compilation unit. Multiple files are implemented with #include files. When using multiple compilation units care must be given that preprocessor commands that control the compilation are compatible across all units. It is recommended directives such as #fuses, #use and the device header file all be 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).

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.     ///
                ////////////////////////////////////////////////////
#if defined(__PCM__)                          // preprocessor directive that
chooses the compiler
#include <16F877.h>                           // preprocessor directive that
selects the chip PIC16F877
#fuses HS,NOWDT,NOPROTECT,NOLVP               // preprocessor directive that
defines fuses for the chip
#use delay(clock=20000000)                    // preprocessor directive that
specifies the clock speed
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7) // preprocessor directive that
includes the rs232 libraries
#elif defined(__PCH__)                        // same as above but for the
PCH compiler and PIC18F452
#include <18F452.h>
#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay(clock=20000000)
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7)
#endif
void main() {                                 // main function
   int i, value, min, max;                    // local variable declaration
   printf("Sampling:");                       // printf function included in
the RS232 library
   setup_port_a( ALL_ANALOG );                // A/D setup functions- built-
in
   setup_adc( ADC_CLOCK_INTERNAL );           // A/D setup functions- built-
in
   set_adc_channel( 0 );                      // A/D setup functions- built-
in
   do {                                       // do while statement
      min=255;                                // expression
      max=0;
      for(i=0; i<=30; ++i) {                  // for statement
         delay_ms(100);                       // delay built-in function
call
         value = Read_ADC();                  // A/D read functions- built-
in
         if(value<min)                        // if statement
            min=value;
         if(value>max)                        // if statement
            max=value;
      }
      printf("\n\rMin: %2X  Max: %2X\n\r",min,max);
   } while (TRUE);
}
```

STATEMENTS

Statements

STATEMENT	

if (expr) stmt; [else stmt;]	`if (x==25)`
	`  x=1;`
	`else`
	`  x=x+1;`

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

do stmt while (expr);	`do {`
	`  putc(c=getc());`
	`} while (c!=0);`

for (expr1;expr2;expr3) stmt;	`for (i=1;i<=10;++i)` `  printf("%u\r\n",i);`

switch (expr) {
case cexpr: stmt; //one or more case
[default:stmt]
... }
	`switch (cmd) {`
	`  case 0: printf("cmd 0");`
	`  break;`
	`  case 1: printf("cmd 1");`
	`  break;`
	`  default: printf("bad cmd");`
	`  break; }`

return [expr];	`return (5);`
goto label;	`goto loop;`
label: stmt;	`loop: I++;`
break;	`break;`
continue;	`continue;`
expr;	`i=1;`
;	`;`
{[stmt]} Zero or more	`{a=1;` `  b=1;}`

Note: Items in [ ] are optional

EXPRESSIONS

Expressions

Constants:
123
0123
0x123
0b010010
'x'
'\010'
'\xA5'
'\c'
"abcdef"

Identifiers:	
ABCDE	Up to 32 characters beginning with a non-numeric. Valid characters are A-Z, 0-9 and _ (underscore).
ID[X]	Single Subscript
ID[X][X]	Multiple Subscripts
ID.ID	Structure or union reference
ID->ID	Structure or union reference

Operators

+	Addition Operator
+=	Addition assignment operator, x+=y, is the same as x=x+y
&=	Bitwise and assignment operator, x&=y, is the same as x=x&y

&	Address operator
&	Bitwise and operator
^=	Bitwise exclusive or assignment operator, x^=y, is the same as x=x^y
^	Bitwise exclusive or operator
l=	Bitwise inclusive or assignment operator, xl=y, is the same as x=xly
l	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
>=	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
<=	Less than or equal to operator
&&	Logical AND operator
!	Logical negation operator
ll	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
-	Subtraction operator
sizeof	Determines size in bytes of operand

Operator Precedence

IN DESCENDING PRECEDENCE					
(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			

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 Parameters

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 paremter and calls the proper function.

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

This function finds the square root of a long integer variable (from the pointer), saves result back to pointer.

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

This function finds the square root of a float variable (from the pointer), saves result back to pointer.

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.

```
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 CCS C all the variables should be declared before it is used. They can be defined inside a function (local) or outside all functions (global). This would affect the visibility and life of the variables.

Basic Types

Type-Specifier	
int1	Defines a 1 bit number
int8	Defines an 8 bit number
int16	Defines a 16 bit number
int32	Defines a 32 bit number
char	Defines a 8 bit character
float	Defines a 32 bit floating point number
short	By default the same as int1
Int	By default the same as int8
long	By default the same as int16
void	Indicates no specific type

Note: All types, except float, by default are unsigned; however, maybe preceded by unsigned or signed. Short and long may have the keyword INT following them with no effect. Also see #TYPE to change the default size.

SHORT is a special type used to generate very efficient code for bit operations and I/O. Arrays of bits (INT1 or SHORT) in RAM are now supported. Pointers to bits are not permitted.

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	Is allowed as a qualifier however, has no effect.
_fixed(n)	Creates a fixed point decimal number where *n* is how many decimal places to round to.

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 e.g.:

```
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 cexpr [cexpr]]]}

One or more semi-colon separated **Zero or more**

For e.g.:

```
struct data_record  {
int   a [2];
int  b : 2; /*2 bits */
int  c : 3; /*3  bits*/
int d;
};                              // data_record is a structure type
```

Union 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 cexpr [cexpr]]]}

One or more semi-colon separated

Zero or more

For e.g.:

```
union u_tag {
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 eg:

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

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

For eg:

```
___ADDRESS__ testa = 0x1000  //will allocate 16 bits for testa 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 e.g.:

```
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 eg:

```
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);
```

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){

}
```

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,DataEEread,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 eg:

```
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. The syntax is const type specifier id [cexpr] ={value}
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.

For e.g.:

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

For placing a string into ROM

```
const char cstaring[6]={"hello"}
```

You can also create pointers to constants

```
const char *cptr;
cptr = string;
```

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

For eg:

#ORG 0x1C00, 0x1C0F
CONST CHAR ID[10]= {"123456789"};
This ID will be at 1C00.

Note: some extra code will proceed the 123456789.

A new method allows the use of pointers to ROM. The new keyword for compilation modes CCS4 and ANSI is ROM and for other modes it is _ROM. This method does not contain extra code at the start of the structure.
For e.g:

```
char rom commands[] = {“put|get|status|shutdown”};
```

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.

For PIC18 parts the compiler allows a non-standard c feature to implement a constant array of variable length strings. The syntax is:

```
const char id[n] [*] = { "strint", "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 usage is #rom address ={data, data,..,data}.

For eg:

```
#rom 0x1000={1,2,3,4,5}  // will place 1,2,3,4,5 to rom addresses
starting at 0x1000
```

This can be used for strings #rom address={"hello"} // the string will be null terminated. 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_eeprom(address,data)*- writes 16 bit data to program memory
- *write_program_memory(address, dataptr, count)*; writes count bytes of data from dataptr to address in program memory.

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 all the three methods above can be read form user code using:

- *read_program_eeprom(address)- r*eads 16 bits data from the address in program memory.
- *read_program_memory((address, dataptr, count)* -Reads count bytes from program memory at 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.

FUNCTIONAL OVERVIEWS

I2C

I²C: These options lets the user configure the hardware to communicate with other devices over the I²C interface. All devices do not have hardware I²C. Software I²C can be used for these devices.

Relevant Functions:

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

Relevant Preprocessor:

#use i²c	Configures the device as a Master or a Slave and assigns the SDA and SCL pins used for the interface.

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)

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)  // Configure Device as Master
..
..
BYTE data;                  // Data to be transmitted

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

ADC

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

Relevant Functions:

setup_adc(mode)	Sets up the a/d mode like off, the adc clock etc.
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.

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_AD	Interrupt fires when a/d conversion is complete
INT_ADOF	Interrupt fires when a/d conversion has timed out

Relevant Include Files:

None, all functions built-in

Relevant getenv() parameters:

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

Example Code:

```
  #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
                                  If the device had a 10bit ADC module,  value will range
                                  between 0-3FF. #DEVICE ADC=8 will yield 0-FF and
                                  #DEVICE ADC=16 will yield 0-FFC0
```

Analog Comparator

These functions sets 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, please 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 hence more interrupts.

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:

COMP	Returns 1 if the device has comparator

Example Code:
For eg:

```
 For PIC12F675
   setup_adc_ports(NO_ANALOGS); // all pins digital
   setup_comparator(A0_A1_OUT_ON_A2);//a0 and a1 are analog comparator inputs and a2 is the
                                                    // output if(C1OUT) //true when comparator output is
                                                     high output_low(pin_a4); else output_high(pin_a4);
```

CAN Bus

These functions allow easy access to the Controller Area Network (CAN) features included with the MCP2515 CAN interface chip and the PIC18 MCU. 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 work ECAN at the end of the description. The listed interrupts are no available to the MCP2515 interface chip.

Relevant Functions:

can_init(void);	Initializes the CAN module to 125k baud 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. 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 first out (fifo) mode. Only available on chips with ECAN modules.
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.
```

CCP1

These options allow for configuration and use of the CCP module. There might be multiple CCP modules for a device. These functions are only available on devices with CCP hardware. They operate in 3 modes: capture, compare and PWM. The source in capture/compare mode can be timer1 or timer3 and in PWM can be timer2 or timer4. The options available are different for different devices and are listed in the device header file. In capture mode the value of the timer is copied to the CCP_X register when the input pin event occurs. In compare mode it will trigger an action when timer and CCP_x values are equal and in PWM mode it will generate a square wave.

Relevant Functions:

setup_ccp1(mode)	Sets the mode to capture, compare or PWM. For capture
set_pwm1_duty(value)	The value is written to the pwm1 to set the duty.

Relevant Preprocessor:
None

Relevant Interrupts :

INT_CCP1	Interrupt fires when capture or compare on CCP1

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:

CCP1	Returns 1 if the device has CCP1

Example Code:

```
  #int_ccp1
 void isr()
 {
    rise = CCP_1;                    //CCP_1 is the time the pulse went high
    fall = CCP_2;                    //CCP_2 is the time the pulse went low
    pulse_width = fall - rise;       //pulse width
 }

..
 setup_ccp1(CCP_CAPTURE_RE);         // Configure CCP1 to capture rise
 setup_ccp2(CCP_CAPTURE_FE);         // Configure CCP2 to capture fall
 setup_timer_1(T1_INTERNAL);         // Start timer 1
```

Some chips also have fuses which allows to multiplex the ccp/pwm on different pins. Be sure to check the fuses to see which pin is set by default. Also fuses to enable/disable pwm outputs.

CCP2, CCP3, CCP4, CCP5, CCP6

Similar to CCP1

Configuration Memory

On all PIC18 MCUs, the configuration memory is readable and writable. This functionality is not available on 14-bit MCUs.

Relevant Functions:

write_configuration_memory (ramaddress, count) or	Writes count bytes, no erase needed
write_configuration_memory (offset,ramaddress, count)	Writes count bytes, no erase needed starting at byte address offset
read_configuration_memory (ramaddress,count)	Read count bytes of configuration memory

Relevant Preprocessor:
None

Relevant Interrupts :
None

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:
None

Example Code:

```
For PIC18f452
 int16 data=0xc32;
 ...
 write_configuration_memory(data,2);//writes 2 bytes to the configuration memory
```

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:

read_eeprom(address)	Reads the data eeprom memory location(8 bit or 16 bit depending on the device).
write_eeprom(address, value)	Erases and writes value(8 bit) to data eeprom location address.

Relevant Preprocessor:

#ROM address={list}	Can also be used to put data eeprom memory data into the hex file.

Relevant Interrupts :

INT_EEPROM	Interrupt fires when eeprom write is complete

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:

DATA_EEPROM	Size of data eeprom memory.

Example Code:
For eg: For 18F452

```
#rom 0xf00000={1,2,3,4,5}  //inserts this data into the hex file. The data eeprom address differs
                           //for different family of chips. Please refer to the programming
                           //specs to find the right value for the device.

write_eeprom(0x0,0x12);    //writes 0x12 to  data eeprom location 0
value=read_eeprom(0x0);    //reads data eeprom location 0x0 returns 0x12
```

External Memory

Some PIC 18 MCUs have the external memory functionality where the external memory can be mapped to external memory devices like(Flash, EEPROM or RAM). These functions are available only on devices that support external memory bus.

Relevant Functions:

setup_external_memory(mode)	Sets the mode of the external memory bus refer to the device header file for available constants.
read_external_memory (address, dataptr,count)	Reads count bytes to dataptr form address.
write_external_memory (address_dataptr,count)	Writes count bytes from dataptr to address

These functions do not use any Flash/EEPROM write algorithm. The data is only copied to/from register data address space to/from program memory address space.

Relevant Preprocessor:
None

Relevant Interrupts :
None

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:
None

Example Code:

```
write_external_memory(0x20000,data,2);    //writes2 bytes form data to 0x20000(starting
                                          // address of external memory)

read_external_memory(0x20000,value,2)     //reads 2 bytes from 0x20000 to value
```

Internal LCD

Some families of PIC® microcontrollers can drive an LCD glass directly, without the need of an LCD controller. For example, the PIC16C926, PIC16F916 and the PIC18F8490 have an internal LCD controller.

Relevant Functions:

setup_lcd (mode, prescale, segments)	Configures the LCD module to use the specified segments specified mode and specified timer prescalar. For more information on valid modes see the setup_lcd() manual page and the .H header file for your PICmicro controller.
lcd_symbol (symbol, segment_b7 .. segment_b0)	The specified symbol is placed on the desired segments. For example, if bit0 of symbol is set, then segment_b0 is set. Segment_b7 to segment_b0 represent the SEGXX pin on the PICmicro. In this example, if bit0 of symbol is set and segment_b0 is 15, then SEG15 would be set.
lcd_load(ptr, offset, len)	Writes len bytes of data from ptr directly to the LCD segment memory, starting with offset.

Relevant Preprocessor:
None

Relevant Interrupts:

#int_lcd	LCD frame is complete, all pixels displayed

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

Relevant getenv() Parameters:

LCD	Returns TRUE if the device has an internal LCD controller.

Example Program:

```
//how each segment is set (on or off) for ascii digits 0 to 9.
byte CONST DIGIT_MAP[10]={0X90,0XB7,0X19,0X36,0X54,0X50,0XB5,0X24};

//define the segment information for the 1st digit of the glass LCD.
//in this example the first segment uses the second seg signal on COM0
#define DIGIT_1_CONFIG  COM0+2,COM0+4,COM0+5,COM2+4,COM2+1, COM1+4,COM1+5

//display digits 1 to 9 on the first digit of the LCD
for(i=1; i<=9; ++i) {
        LCD_SYMBOL(DIGIT_MAP[i],DIGIT_1_CONFIG);
        delay_ms(1000);
}
```

Internal Oscillator

Many chips have an internal oscillator. There are different ways to configure the internal oscillator. Some chips have a constant 4 Mhz factory calibrated internal oscillator. The value is stored in some location (usually the highest program memory) and the compiler moves it to the osccal register on startup. The programmers save and restore this value but if this is lost they need to be programmed before the oscillator is functioning properly. Some chips have a factory calibrated internal oscillator that offers a software selectable frequency range(from 31Kz to 8 Mhz). They have a default value and can be switched to a higher/lower value in software, as well as software tunable. Some chips also provide the PLL option for the internal oscillator.

Relevant Functions:

setup_oscillator (mode, finetune)	Sets the value of the internal oscillator and also tunes it. The options vary depending on the chip and are listed in the device header files.

Relevant Preprocessor:
None

Relevant Interrupts:

INT_OSC_FAIL or INT_OSCF	Interrupt fires when the system oscillator fails and the processor switches to the internal oscillator.

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:
None

Example Code:
For PIC18F8722
setup_oscillator(OSC_32MHZ);//sets the internal oscillator to 32MHz (PLL enabled)
If the internal oscillator fuse option is specified in the #fuses and a valid clock is specified in the #use delay(clock=xxx) directive the compiler automatically sets up the oscillator. The #use delay statements should be used to tell the compiler about the oscillator speed.

Interrupts

The following functions allow for the control of the interrupt subsystem of the microcontroller. With these functions, interrupts can be enable, 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 the specified interrupt flag. This can be used if a global isr is used, or to prevent an interrupt from being serviced.

Relevant Preprocessor:

#device high_ints= This directive tells the compiler to generate code for high priority interrupts.
#int_xxx fast This directive tells the compiler that the specified interrupt should be treated as a high priority interrupt.

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 complier 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_interrtups(TIMER0);     // disables the timer0 interrupt
clear_interrupt(TIMER0);        // clears the timer0 interrupt flag
```

Low Voltage Detect

These functions configure the high/low voltage detect module. Functions available on the chips that have the low voltage detect hardware.

Relevant Functions:

setup_low_volt_detect(mode)	Sets the voltage trigger levels and also the mode(below or above in case of the high/low voltage detect module). The options vary depending on the chip and are listed in the device header files.

Relevant Preprocessor:
None

Relevant Interrupts :

INT_LOWVOLT	Interrupt fires on low voltage detect

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:
None

Example Code:
For PIC18F8722

```
setup_low_volt_detect(LVD_36|LVD_TRIGGER_ABOVE);   //sets the trigger level as 3.6 volts
                                                   //and trigger direction as above.
                                                   //The interrupt if enabled is fired
                                                   //when the voltage is above 3.6
                                                   //volts.
```

Power PWM

These options lets the user configure the Pulse Width Modulation (PWM) pins. They are only available on devices equipped with PWM. The options for these functions vary depending on the chip and are listed in the device header file.

Relevant Functions:

setup_power_pwm(config)	Sets up the PWM clock, period, dead time etc.
setup_power_pwm_pins (module x)	Configure the pins of the PWM to be in Complimentary, ON or OFF mode.
set_power_pwmx_duty(duty)	Stores the value of the duty cycle in the PDCXL/H register. This duty cycle value is the time for which the PWM is in active state.
set_power_pwm_override (pwm,override,value)	This function determines whether the OVDCONS or the PDC registers determine the PWM output .

Relevant Preprocessor:
None.

Relevant Interrupts:

#INT_PWMTB	PWM Timebase Interrupt (Only available on PIC18XX31)

Relevant getenv() Parameters:
None.

Example Code:

```
....
long duty_cycle, period;
...
                    // Configures PWM  pins to be ON,OFF or in Complimentary mode.
setup_power_pwm_pins(PWM_COMPLEMENTARY ,PWM_OFF, PWM_OFF, PWM_OFF);

                    //Sets up PWM clock , postscale and period. Here period is used to set the
                    //PWM Frequency as follows:
                    //Frequency = Fosc / (4 * (period+1) *postscale)
setup_power_pwm(PWM_CLOCK_DIV_4|PWM_FREE_RUN,1,0,period,0,1,0);

set_power_pwm0_duty(duty_cycle)); // Sets the duty cycle of the PWM 0,1 in
                                  //Complementary mode
```

Program EEPROM

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

Relevant Functions:

read_program_eeprom (address)	Reads the program memory location(16 bit or 32 bit depending on the device).
write_program_eeprom (address, value)	Writes value to program memory location address.
erase_program_eeprom (address)	Erases FLASH_ERASE_SIZE bytes in program memory.
write_program_memory (address,dataptr,count)	Writes count bytes to program memory from dataptr to address. When address is a mutiple of FLASH_ERASE_SIZE an erase is also performed.
read_program_memory (address,dataptr,count)	Read count bytes from program memory at address to dataptr.

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:
For 18F452 where the write size is 8 bytes and the erase size is 64 bytes

```
#rom 0xa00={1,2,3,4,5}                  //inserts this data into the hex file.
erase_program_eeprom(0x1000);           //erases 64 bytes strting at 0x1000
write_program_eeprom(0x1000,0x1234);    //writes the data 0x1234 to the address
                                        //0x1000
value=read_program_eeprom(0x1000);      //reads from 0x1000 returns 0x1234
write_program_memory(0x1000,data,8);    //erases 64 bytes starting at 0x1000 as
                                        //0x1000 is a multiple of 64 and writes 8 bytes
                                        //from data to 0x1000
read_program_memory(0x1000,value,8);    //reads 8 bytes to value from 0x1000
erase_program_eeprom(0x1000);           //erases 64 bytes starting at 0x1000
write_program_memory(0x1000,data,8);    //writes 8 bytes from data to 0x1000
read_program_memory(0x1000,value,8);    //reads 8 bytes to value from 0x1000
```

For chips where getenv("FLASH_ERASE_SIZE") > getenv("FLASH_WRITE_SIZE")
WRITE_PROGRAM_EEPROM - Writes 2 bytes, does not erase (use PROGRAM_EEPROM)
WRITE_PROGRAM_MEMORY - Writes any number of bytes,
will erase a block whenever the first (lowest) byte
in a block is written to. If the first address is
not the start of a block that block is not erased.
ERASE_PROGRAM_EEPROM - Will erase a block. The lowest address bits are not used.

For chips where getenv("FLASH_ERASE_SIZE") = getenv("FLASH_WRITE_SIZE")
WRITE_PROGRAM_EEPROM - Writes 2 bytes, no erase is needed.
WRITE_PROGRAM_MEMORY - Writes any number of bytes, bytes outside the range
of the write block are not changed. No erase is needed.
ERASE_PROGRAM_EEPROM - Not available.

PSP

These options allow for configuration and use of the Parallel Slave Port on the supported devices.

Relevant Functions:

setup_psp(mode)	Enables/disables the psp port on the chip
psp_output_full()	Returns 1 if the output buffer is full(waiting to be read by the external bus)
psp_input_full	Returns 1 if the input buffer is full(waiting to read by the cpu)
psp_overflow	Returns 1 if a write occurred before the previously written byte was read

Relevant Preprocessor:
None

Relevant Interrupts :

INT_PSP	Interrupt fires when PSP data is in

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:

PSP	Returns 1 if the device has PSP

Example Code:
For eg:

```
  while(psp_output_full());          //waits till the output buffer is cleared
  psp_data=command;                  //writes to the port
  while(!input_buffer_full());       //waits till input buffer is cleared
  if (psp_overflow())
    error=true                       //if there is an overflow set the error flag
  else
    data=psp_data;                   //if there is no overflow then read the port
```

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.
PUTC or PUTCHAR or FPUTC	Puts a character over the transmit pin(on the specified stream in the case of FPUTC, stdout by default)
PUTS or FPUTS	Puts a string over the transmit pin(on the specified stream in the case of FPUTC, stdout by default). Uses putc to send each character.
PRINTF 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
KBHIT	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.

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:
None

Example Code:
For eg:
For PIC16F877

```
 #use rs232(baud=9600, xmit=pin_c6,rcv=pin_c7)
printf("enter a character");
if (kbhit())
return getc();
```

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 to semaphore to claim the resource.
rtos_await(expre)	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 over ran its alloted 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 it's 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.
```

SPI

Most PIC® microcontrollers have an internal Serial Parallel Interface (or SPI) peripheral. SPI is a 3 or 4 wire synchronous serial connection. CCS provides built-in functions to control the internal SPI peripheral.

Relevant Functions:

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

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

spi_data_is_in() spi_data_is_in2()	Returns TRUE if the SPI receive buffer has a byte of data.
spi_write(value) spi_write2(value)	Transmits the value over the SPI interface. This will cause the data to be clocked out on the SDO pin.
spi_read(value) spi_read2(value)	Performs an SPI transaction, where the value is clocked out on the SDO pin and data clocked in on the SDI pin is returned. If you just want to clock in data then you can use spi_read() without a parameter**.**

Relevant Preprocessor:
None

Relevant Interrupts:

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

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.
```

Timer0

These options allow the user configure and use timer0. It is available on all devices and is always enabled. The clock/counter is 8-bit on PIC16 microcontrollers and 8 or 16-bit on PIC 18 MCUs. It counts up and also provides interrupt on overflow. The options available differ and are listed in the device header file.

Relevant Functions:

setup_timer_0(mode)	Sets the source, prescale etc for timer0
set_timer0(value) or set_rtcc(value)	Initializes the timer0 clock/counter. Value may be a 8-bit or 16-bit depending on the device
value=get_timer0	Returns the value of the timer0 clock/counter

Relevant Preprocessor:
None

Relevant Interrupts :

INT_TIMER0 or INT_RTCC	Interrupt fires when timer0 overflows

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:

TIMER0	Returns 1 if the device has timer0

Example Code:

```
  For PIC18F452
  setup_timer0(RTCC_INTERNAL|RTCC_DIV_2|RTCC_8_BIT);//sets the internal clock as source
                                                    //and prescale 2. At 20Mhz timer0
                                                    //will increment every .4us in this
                                                    //setup and overflows every
                                                    //1.024ms
  set_timer0(0);                                   //this sets timer0 register to 0
  time=get_timer0();                               //this will read the timer0 register
                                                    //value
```

Timer1

These options lets the user configure and use timer1. The clock/counter is 16-bit on PIC 16 and PIC 18 MCUs. It counts up and also provides interrupt on overflow. The options available differ and are listed in the device header file.

Relevant Functions:

setup_timer_1(mode)	Disables or sets the source and prescale for timer1
set_timer1(value)	Initializes the timer1 clock/counter
value=get_timer1	Returns the value of the timer1 clock/counter

Relevant Preprocessor:
None

Relevant Interrupts :

INT_TIMER1	Interrupt fires when timer0 overflows

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:

TIMER1	Returns 1 if the device has timer1

Example Code:

```
  For PIC18F452
  setup_timer1(T1_DISABLED);                    //disables timer1
     or
  setup_timer1(T1_INTERNAL|T1_DIV_BY_8);//sets the internal clock as source
                                                //and prescale as 8. At 20Mhz timer1 will increment
                                                //every 1.6us in this setup and overflows every
                                                //104.896ms
  set_timer1(0);                               //this sets timer1 register to 0
  time=get_timer1();                           //this will read the timer1 register value
```

Timer2

These options lets the user configure and use timer2. The clock/counter is 8-bit on PIC 16 and PIC 18 MCUs. It counts up and also provides interrupt on overflow. The options available differ and are listed in the device header file.

Relevant Functions:

setup_timer_2 (mode,period,postscale)	Disables or sets the prescale, period and a postscale for timer2
set_timer2(value)	Initializes the timer2 clock/counter
value=get_timer2	Returns the value of the timer2 clock/counter

Relevant Preprocessor:
None

Relevant Interrupts :

INT_TIMER2	Interrupt fires when timer2 overflows

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:

TIMER2	Returns 1 if the device has timer2

Example Code:

```
For PIC18F452
setup_timer2(T2_DISABLED);            //disables timer2
     or
setup_timer2(T2_DIV_BY_4,0xc0,2);//sets the prescale as 4, period as 0xc0 and postscales as 2.
                                  //At 20Mhz timer2 will increment every .8us in this
                                  //setup overflows every 154.4us and interrupt every 308.2us
 set_timer2(0);                   //this sets timer2 register to 0
 time=get_timer2();               //this will read the timer1 register value
```

Timer3

Timer3 is very similar to timer1. So please refer to the Timer1 section for more details.

Timer4

Timer4 is very similar to timer2. So please refer to the Timer2 section for more details.

Timer5

These options lets the user configure and use timer5. The clock/counter is 16-bit and is available only on 18Fxx31 devices. It counts up and also provides interrupt on overflow. The options available differ and are listed in the device header file.

Relevant Functions:

setup_timer_5(mode)	Disables or sets the source and prescale for timer5
set_timer5(value)	Initializes the timer5 clock/counter
value=get_timer5	Returns the value of the timer51 clock/counter

Relevant Preprocessor:
None

Relevant Interrupts :

INT_TIMER5	Interrupt fires when timer5 overflows

Relevant Include Files:
None, all functions built-in

Relevant getenv() parameters:

TIMER5	Returns 1 if the device has timer5

Example Code:

```
For PIC18F4431
setup_timer5(T5_DISABLED)                    //disables timer5
      or
setup_timer1(T5_INTERNAL|T5_DIV_BY_1); //sets the internal clock as source and prescale as 1.
                                       //At 20Mhz timer5 will increment every .2us in this
                                       //setup and overflows every 13.1072ms
  set_timer5(0);                       //this sets timer5 register to 0
  time=get_timer5();                   //this will read the timer5 register value
```

USB

Universal Serial Bus, or USB, is used as a method for peripheral devices to connect to and talk to a personal computer. CCS provides libraries for interfacing a PIC to PC using USB by using a PIC with an internal USB peripheral (like the PIC16C765 or the PIC18F4550 family) or by using any PIC with an external USB peripheral (the National USBN9603 family).

Relevant Functions

usb_init()	Initializes the USB hardware. Will then wait in an infinite loop for the USB peripheral to be connected to bus (but that doesn't mean it has been enumerated by the PC). Will enable and use the USB interrupt.
usb_init_cs()	The same as usb_init(), but does not wait for the device to be connected to the bus. This is useful if your device is not bus powered and can operate without a USB connection.
usb_task()	If you use connection sense, and the usb_init_cs() for initialization, then you must periodically call this function to keep an eye on the connection sense pin. When the PIC is connected to the BUS, this function will then prepare the USB peripheral. When the PIC is disconnected from the bus, it will reset the USB stack and peripheral. Will enable and use the USB interrupt.

Note: In your application you must define USB_CON_SENSE_PIN to the connection sense pin.

usb_detach()	Removes the PIC from the bus. Will be called automatically by usb_task() if connection is lost, but can be called manually by the user.
usb_attach()	Attaches the PIC to the bus. Will be called automatically by usb_task() if connection is made, but can be called manually by the user.
usb_attached()	If using connection sense pin (USB_CON_SENSE_PIN), returns TRUE if that pin is high. Else will always return TRUE.
usb_enumerated()	Returns TRUE if the device has been enumerated by the PC. If the device has been enumerated by the PC, that means it is in normal operation mode and you can send/receive packets.
usb_put_packet (endpoint, data, len, tgl)	Places the packet of data into the specified endpoint buffer. Returns TRUE if success, FALSE if the buffer is still full with the last packet.
usb_puts (endpoint, data, len, timeout)	Sends the following data to the specified endpoint. usb_puts() differs from usb_put_packet() in that it will send multi packet messages if the data will not fit into one packet.
usb_kbhit(endpoint)	Returns TRUE if the specified endpoint has data in its receive buffer.

usb_get_packet (endpoint, ptr, max)	Reads up to max bytes from the specified endpoint buffer and saves it to the pointer ptr. Returns the number of bytes saved to ptr.
usb_gets (endpoint, ptr, max, timeout)	Reads a message from the specified endpoint. The difference between usb_get_packet() and usb_gets() is that usb_gets() will wait until a full message has received, which a message may contain more than one packet. Returns the number of bytes received.

Relevant CDC Functions
A CDC USB device will emulate an RS-232 device, and will appear on your PC as a COM port. The following functions allow for this virtual RS-232/serial interface.

Note: When using the CDC library, you can use the same functions above, but do not use the packet related function such as usb_kbhit(), usb_get_packet(), etc.

usb_cdc_kbhit()	The same as kbhit(), returns TRUE if there is 1 or more character in the receive buffer.
usb_cdc_getc()	The same as getc(), reads and returns a character from the receive buffer. If there is no data in the receive buffer it will wait indefinitely until there a character has been received.
usb_cdc_putc(c)	The same as putc(), sends a character. It actually puts a character into the transmit buffer, and if the transmit buffer is full will wait indefinitely until there is space for the character.
usb_cdc_putc_fast(c)	The same as usb_cdc_putc(), but will not wait indefinitely until there is space for the character in the transmit buffer. In that situation the character is lost.
usb_cdc_putready()	Returns TRUE if there is space in the transmit buffer for another character.

Relevant Preporcessor:
None

Relevant Interrupts:

#int_usb	A USB event has happened, and requires application intervention. The USB library that CCS provides handles this interrupt automatically.

Relevant Include files:

pic_usb.h	Hardware layer driver for the PIC16C765 family PIC® MCUs with an internal USB peripheral.
pic_18usb.h	Hardware layer driver for the PIC18F4550 family PIC® MCUs with an internal USB peripheral.

usbn960x.h	Hardware layer driver for the National USBN9603/USBN9604 external USB peripheral. You can use this external peripheral to add USB to any microcontroller.
usb.h	Common definitions and prototypes used by the USB driver
usb.c	The USB stack, which handles the USB interrupt and USB Setup Requests on Endpoint 0.
usb_cdc.h	A driver that takes the previous include files to make a CDC USB device, which emulates an RS232 legacy device and shows up as a COM port in the MS Windows device manager.

Relevant getenv() Parameters:

USB	Returns TRUE if the PIC® MCU has an integrated internal USB peripheral.

Example Code:

Due to the complexity of USB, example code will not fit here. However, the following examples are included in the CCS C Compiler:

ex_usb_hid.c	A simple HID device
ex_usb_mouse.c	A HID Mouse, when connected to the PC the mouse cursor will go in circles.
ex_usb_kbmouse.c	An example of how to create a USB device with multiple interfaces by creating a keyboard and mouse in one device.
ex_usb_kbmouse2.c	An example of how to use multiple HID report IDs to transmit more than one type of HID packet, as demonstrated by a keyboard and mouse on one device.
ex_usb_scope.c	A vendor-specific class using bulk transfers is demonstrated.
ex_usb_serial.c	The CDC virtual RS232 library is demonstrated with this RS232 < - > USB example.
ex_usb_serial2.c	Another CDC virtual RS232 library example, this time a port of the ex_intee.c example to use USB instead of RS232.

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. The options vary depending on the chip, please refer to the header file for details

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:

setup_wdt	Enables/disables the wdt or sets the prescalar
restart_wdt	Restarts the wdt, if wdt is enables this must be periodically called to prevent a timeout reset.

For 12 and 14-bit chips it is enabled/disabled using WDT or NOWDT fuses, whereas, on PIC 18 devices it is done using the setup_wdt function.
The timeout time for 12 and 14-bit chips are set using the setup_wdt function and on PIC 18 devices
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:

#fuses WDT/NOWDT	Enabled/Disables wdt in 12 and 14-bit devices
#fuses WDT16	Sets ups the timeout time in PIC18 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 are share the WDT prescalar bits with timer0 so the WDT prescalar constants can be used with setup_counters or setup_timer0 or setup_wdt functions.

PRE-PROCESSOR DIRECTIVES

PRE-PROCESSOR

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

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

Examples:
Both of the following are valid

```
#INLINE
#PRAGMA INLINE
```

Standard C	**#DEFINE ID STRING**	**#IF expr**	**#NOLIST**
	#ELSE	**#IFDEF id**	**#PRAGMA cmd**
	#ENDIF	**#INCLUDE "FILENAME"**	**#UNDEF id**
	#ERROR	**#LIST**	
Function Qualifier	**#INLINE**	**#INT_GLOBAL**	**#SEPARATE**
	#INT_DEFAULT	**#INT_xxx**	
Pre-Defined Identifier	**__DATE__**	**__LINE__**	**__PCH__**
	__DEVICE__	**__PCB__**	**__TIME__**
	__FILE__	**__PCM__**	**__FILENAME__**
RTOS	**#TASK**	**#USE RTOS**	

Device Specification	#DEVICE CHIP	#ID "filename"	#FUSES options
	#ID NUMBER	#ID CHECKSUM	#SERIALIZE
Built-in Libraries	#USE DELAY CLOCK	#USE FIXED_IO	#USE RS232
	#USE FAST_IO	#USE 12C	#USE STANDARD_IO
	#USE SPI		
Memory Control	#ASM	#BYTE id=id	#ROM
	#BIT id=id.const	#ENDASM	#TYPE
	#BIT id=const.const	#FILL_ROM	#ZERO_RAM
	#BUILD	#LOCATE id=const	
	#BYTE id=const	#RESERVE	
Compiler Control	#CASE	#OPT n	#PRIORITY
	#IGNORE_WARNINGS	#ORG	

#ASM, #ENDASM

Syntax:	#asm or #asm ASIS ***code*** #endasm
Elements:	***code*** is a list of assembly language instructions
Purpose:	The lines between the #ASM and #ENDASM are treated as assembly code to be inserted. These may be used anywhere an expression is allowed. The syntax is described on the following page. The predefined variable _RETURN_ may be used to assign a return value to a function from the assembly code. Be aware that any C code after the #ENDASM and before the end of the function may corrupt the value. If the second form is used with ASIS then the compiler will not do any automatic bank switching for variables that cannot be accessed from the current bank. The assembly code is used as-is. Without this option the assembly is augmented so variables are always accessed correctly by adding bank switching where needed.

Examples:	`int find_parity (int data)  {`
	
	`int count;`
	`#asm`
	`movlw  0x8`
	`movwf  count`
	`movlw  0`
	`loop:`
	`xorwf  data,w`
	`rrf  data,f`
	`decfsz  count,f`
	`goto  loop`
	`movlw  1`
	`awdwf  count,f`
	`movwf  _return_`
	`#endasm`
	`}`

Example Files:	ex_glint.c
Also See:	None

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
BTFSC f,b	BTFSS f,b
ANDLW k	CALL k
CLRWDT	GOTO k
IORLW k	MOVLW k
RETLW k	SLEEP
XORLW	OPTION
TRIS k	
	14 Bit
	ADDLW k
	SUBLW k
	RETFIE
	RETURN

f may be a constant (file number) or a simple variable
d may be a constant (0 or 1) or W or F
f,b may be a file (as above) and a constant (0-7) or it may be just a bit variable reference.
k may be a constant expression

Note that all expressions and comments are in C like syntax.

PIC 18					
ADDWF	f,d	ADDWFC	f,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	SUBWFB	f,d
SWAPF	f,d	TSTFSZ	f	XORWF	f,d
BCF	f,b	BSF	f,b	BTFSC	f,b
BTFSS	f,b	BTG	f,d	BC	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	+*				

The compiler will set the access bit depending on the value of the file register.

If there is just a variable identifier in the #asm block then the compiler inserts an & before it. And if it is an expression it must be a valid C expression that evaluates to a constant (no & here). In C an un-subscripted array name is a pointer and a constant (no need for &).

#ENDASM

See: #ASM

#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.
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 T0IF = 0xb.2`
	`...`
	`T0IF = 0; // Clear Timer 0 interrupt flag`
	
	`int result;`
	`#bit result_odd = result.0`
	`...`
	`if (result_odd)`

Example Files:	ex_glint.c
Also See:	#byte, #reserve, #locate

#BUILD

Syntax:	#build(***segment*** = ***address***) #build(***segment*** = ***address***, ***segment*** = ***address***) #build(***segment*** = ***start***:***end***) #build(***segment*** = ***start***: ***end***, ***segment*** = ***start***: ***end***) #build(***nosleep***)
Elements:	***segment*** is one of the following memory segments which may be assigned a location: MEMORY, RESET, or INTERRUPT. ***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. ***Nosleep*** is used to prevent the compiler from inserting a sleep at the end of main()
Purpose:	PIC18XXX devices with external ROM or PIC18XXX devices with no internal ROM can direct the compiler to utilize the ROM. When linking multiple compilation units, this directive must appear exactly the same in each compilation unit.

Examples:	`#build(memory=0x20000:0x2FFFF)  //Assigns memory space`
	`#build(reset=0x200,interrupt=0x208) //Assigns start`
	`//location`
	`//of reset and`
	`//interrupt`
	`//vectors`
	`#build(reset=0x200:0x207, interrupt=0x208:0x2ff)`
	`//Assign limited space`
	`//for reset and`
	`//interrupt vectors.`

Example Files:	None
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 = 3`
	`#byte  b_port = 6`
	
	`struct  {`
	`  short int r_w;`
	`  short int c_d;`
	`  int unused : 2;`
	`  int data    : 4; } a_port;`
	`#byte a_port = 5`
	`...`
	`a_port.c_d = 1;`

Example Files:	ex_glint.c
Also See:	#bit, #locate, #reserve

#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 Files:	ex_cust.c
Also See:	None

DATE

Syntax:	__DATE__
Elements:	None
Purpose:	This pre-processor identifier is replaced at compile time with the date of the compile in the form: "31-JAN-03"
Examples:	`printf("Software was compiled on ");` `printf(__DATE__);`
Example Files:	None
Also See:	None

#DEFINE

Syntax:	#define ***id*** text or #define ***id***(***x***,***y***...) text
Elements:	***id*** is a preprocessor identifier, text is any text, ***x***,***y*** 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 id concatenated with the string x. If the text contains a string of the form #idx#idy then parameter idx is concatenated with parameter idy forming a new identifier.

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);`

Example Files:	ex_stwt.c, ex_macro.c
Also See:	#undef, #ifdef, #ifndef

#DEVICE

Syntax: #device ***chip options***
#device ***Compilation mode selection***

Elements: ***Chip Options-***

chip is the name of a specific processor (like: PIC16C74), 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:

*=5	Use 5 bit pointers (for all parts)
*=8	Use 8 bit pointers (14 and 16 bit parts)
*=16	Use 16 bit pointers (for 14 bit parts)
ADC=x	Where x is the number of bits read_adc() should return
ICD=TRUE	Generates code compatible with Microchips ICD debugging hardware.
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.
HIGH_INTS=TRUE	Use this option for high/low priority interrupts on the PIC®18.
%f=.	No 0 before a decimal pint 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. This is the default mode.

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. The pointer size in this mode for PCM and PCH is set to *=16 if the part has RAM over 0FF.
ANSI	Default data type is SIGNED all other modes default is UNSIGNED. Compilation is case sensitive, all other modes are case insensitive. Pointer size is set to *=16 if the part has RAM over 0FF.
CCS2 CCS3	var16 = NegConst8 is compiled as: var16 = NegConst8 & 0xff (no sign extension)Pointer size is set to *=8 for PCM and PCH and *=5 for PCB. The overload keyword is required.

CCS2 only	The default #device ADC is set to the resolution of the part, all other modes default to 8.
	onebit = eightbits is compiled as onebit = (eightbits != 0)
	All other modes compile as: onebit = (eightbits & 1)

Purpose:

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

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

Examples:

Chip Options-

```
#device PIC16C74
#device PIC16C67 *=16
#device *=16  ICD=TRUE
#device PIC16F877 *=16 ADC=10
#device %f=.
printf("%f",.5); //will print .5, without the directive it will
print 0.5
```

Compilation mode selection-

```
#device CCS2  // This will set the ADC to the resolution of the
part
```

Example Files:	ex_mxram.c, ex_icd.c, 16c74.h
Also See:	read_adc()

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 Files:	None
Also See:	#device

#ELIF

See #if expr or #ifdef

#ELSE

See #if expr or #ifdef

#ENDIF

See #if expr or #ifdef

#ERROR

Syntax:	#error ***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 Files:	ex_psp.c
Also See:	None

#EXPORT (options)

Syntax:	#EXPORT (options)
Elements:	***FILE=filname*** The filename which will be generated upon compile. If not given, the filname 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, Linker Overview

__FILE__

Syntax:	__FILE__
Elements:	None
Purpose:	The pre-processor identifier is replaced at compile time with the file path and the filename of the file being compiled.

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

Example Files:	assert.h
Also See:	__line__

__FILENAME__

Syntax:	__FILENAME__
Elements:	None
Purpose:	The pre-processor identifier is replaced at compile time with the filename of the file being compiled.

Examples:	`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:	#fuse ***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 processors allow different levels for certain fuses. To access these levels, assign a value to the fuse. For example, on the 18F452, the fuse PROTECT=6 would place the value 6 into CONFIG5L, protecting code blocks 0 and 3. 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.
Examples:	`#fuses HS,NOWDT`
Example Files:	ex_sqw.c
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
Examples:	#HEXCOMMENT Version 3.1 - only use 876A chips
Example Files:	None
Also See:	None

#ID

Syntax:	#ID ***number 16*** #ID ***number***, ***number***, ***number***, ***number*** #ID ***"filename"*** #ID ***CHECKSUM***
Elements:	***Number16*** is a 16 bit number, ***number*** is a 4 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 16-bit number and put one nibble in each of the four ID words in the traditional manner. The second syntax specifies the exact value to be used in each of the four ID words. 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  0x1234`
	`#id  "serial.num"`
	`#id  CHECKSUM`

Example Files:	ex_cust.c
Also See:	None

#ID CHECKSUM

See #ID

#ID "filename"

See #ID

#ID number 16

See #ID

#ID number, number, number, number

See #ID

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

Example Files:	ex_sqw.c
Also See:	#if

#IGNORE_WARNINGS

Syntax:	#ignore_warnings ALL #IGNORE_WARNINGS NONE #IGNORE_WARNINGS ***warnings***
Elements:	***warnings*** is one or more warning numbers separated by commas
Purpose:	This function will suppress warning messages from the compiler. ALL indicates no warning will be generated. NONE indicates all warnings will be generated. If numbers are listed then those warnings are suppressed.

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

Example Files:	None
Also See:	Warning messages

#IMPORT (options)

Syntax:	#Import (options)
Elements:	***FILE=filname*** 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.
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, Linker Overview

#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:	PCW IDE

#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_AD	Analog to digital conversion complete
#INT_ADOF	Analog to digital conversion timeout
#INT_BUSCOL	Bus collision
#INT_BUTTON	Pushbutton
#INT_CANERR	An error has occurred in the CAN module
#INT_CANIRX	An invalid message has occurred on the CAN bus
#INT_CANRX0	CAN Receive buffer 0 has received a new message
#INT_CANRX1	CAN Receive buffer 1 has received a new message
#INT_CANTX0	CAN Transmit buffer 0 has completed transmission
#INT_CANTX1	CAN Transmit buffer 0 has completed transmission
#INT_CANTX2	CAN Transmit buffer 0 has completed transmission
#INT_CANWAKE	Bus Activity wake-up has occurred on the CAN bus
#INT_CCP1	Capture or Compare on unit 1
#INT_CCP2	Capture or Compare on unit 2
#INT_CCP3	Capture or Compare on unit 3
#INT_CCP4	Capture or Compare on unit 4
#INT_CCP5	Capture or Compare on unit 5
#INT_COMP	Comparator detect
#INT_COMP1	Comparator 1 detect
#INT_COMP2	Comparator 2 detect
#INT_CR	Cryptographic activity complete
#INT_EEPROM	Write complete
#INT_EXT	External interrupt
#INT_EXT1	External interrupt #1
#INT_EXT2	External interrupt #2
#INT_EXT3	External interrupt #3
#INT_I2C	I2C interrupt (only on 14000)
#INT_IC1	Input Capture #1
#INT_IC2	Input Capture #2
#INT_IC3	Input Capture #3
#INT_LCD	LCD activity
#INT_LOWVOLT	Low voltage detected
#INT_LVD	Low voltage detected

#INT_OSC_FAIL	System oscillator failed
#INT_OSCF	System oscillator failed
#INT_PSP	Parallel Slave Port data in
#INT_PWMTB	PWM Time Base
#INT_RA	Port A any change on A0_A5
#INT_RB	Port B any change on B4-B7
#INT_RC	Port C any change on C4-C7
#INT_RDA	RS232 receive data available
#INT_RDA0	RS232 receive data available in buffer 0
#INT_RDA1	RS232 receive data available in buffer 1
#INT_RDA2	RS232 receive data available in buffer 2
#INT_RTCC	Timer 0 (RTCC) overflow
#INT_SPP	Streaming Parallel Port Read/Write
#INT_SSP	SPI or I2C activity
#INT_SSP2	SPI or I2C activity for Port 2
#INT_TBE	RS232 transmit buffer empty
#INT_TBE0	RS232 transmit buffer 0 empty
#INT_TBE1	RS232 transmit buffer 1 empty
#INT_TBE2	RS232 transmit buffer 2 empty
#INT_TIMER0	Timer 0 (RTCC) overflow
#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_USB	Universal Serial Bus activity

Note many more #INT_ options are available on specific chips. Check the devices .h file for a full list for a given chip.

Elements: None

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 compiler will generate code to jump to the function when the interrupt is detected. It 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 along with the ENABLE_INTERRUPTS(GLOBAL) to enable interrupts.

The keywords HIGH and FAST may be used with the PCH compiler to mark an interrupt as high priority. A high-priority interrupt can interrupt another interrupt handler. An interrupt marked FAST is performed without saving or restoring any registers. You should do as little as possible and save any registers that need to be saved on your own. Interrupts marked HIGH can be used normally. See #DEVICE for information on building with high-priority interrupts.

A summary of the different kinds of PIC18 interrupts:

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

#INT_xxxx FAST
High priority interrupt. Compiler DOES NOT save/restore key registers. This interrupt will interrupt any normal interrupt in progress. Only one is allowed in a program.

#INT_xxxx HIGH
High priority interrupt. Compiler saves/restores key registers. This interrupt will interrupt any normal interrupt in progress.

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

Examples:

```
#int_ad
adc_handler() {
   adc_active=FALSE;
}

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

Example Files: See ex_sisr.c and ex_stwt.c for full example programs.

Also See: enable_interrupts(), disable_interrupts(), #int_default, #int_global, #PRIORITY

#INT_DEFAULT

Syntax:	#int_default
Elements:	None
Purpose:	The following function will be called if the PIC® triggers an interrupt and none of the interrupt flags are set. If an interrupt is flagged, but is not the one triggered, the #INT_DEFAULT function will get called.

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

Example Files:	None
Also See:	#INT_xxxx, #INT_global

#INT_GLOBAL

Syntax:	#int_global
Elements:	None
Purpose:	This directive causes the following function to replace the compiler interrupt dispatcher. The function is normally not required and should be used with great caution. When used, the compiler does not generate start-up code or clean-up code, and does not save the registers.

Examples:	`#int_global`
	`isr() {  // Will be located at location 4 for PIC16 chips.`
	`  #asm`
	`  bsf  isr_flag`
	`  retfie`
	`  #endasm`
	`}`

Example Files:	ex_glint.c
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

#LOCATE

Syntax:	#locate ***id***=***x***
Elements:	***id*** is a C variable, ***x*** is a constant memory address
Purpose:	#LOCATE works like #BYTE however in addition it prevents C from using the area.

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=0x50`

Example Files:	ex_glint.c
Also See:	#byte, #bit, #reserve

#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 #included 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 #defines can be given scope, which cannot normally be done in standard C methodology.

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

Example Files:	None
See Also:	#EXPORT, #MODULE, Invoking the Command Line Compiler, Linker Overview

#NOLIST

Syntax:	#nolist
Elements:	None
Purpose:	Stops inserting source lines into the .LST file (until a #LIST)

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

Example Files:	16c74.h
Also See:	#LIST

#OPT

Syntax:	#OPT ***n***
Elements:	All Devices: n is the optimization level 0-9 PIC18XXX: ***n*** is the optimization level 0-11
Purpose:	The optimization level is set with this directive. This setting applies to the entire program and may appear anywhere in the file. Optimization level 5 will set the level to be the same as the PCB, PCM, and PCH standalone compilers. The PCW default is 9 for full optimization. PIC18XXX devices may utilize levels 10 and 11 for extended optimization. Level 9 may be used to set a PCW compile to look exactly like a PCM compile for example. It may also be used if an optimization error is suspected to reduce optimization.
Examples:	`#opt 5`
Example Files:	None
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 or constant 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'ed 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'ed 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.

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 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

__PCB__

Syntax:	__PCB__
Elements:	None
Purpose:	The PCB compiler defines this pre-processor identifier. It may be used to determine if the PCB compiler is doing the compilation.

Examples:	`#ifdef __pcb__`
	`#device PIC16c54`
	`#endif`

Example Files:	ex_sqw.c
Also See:	__PCM__, __PCH__

__ PCM __

Syntax:	__PCM__
Elements:	None
Purpose:	The PCM compiler defines this pre-processor identifier. It may be used to determine if the PCM compiler is doing the compilation.

Examples:	`#ifdef __pcm__`
	`#device PIC16c71`
	`#endif`

Example Files:	ex_sqw.c
Also See:	__PCB__, __PCH__

__ PCH __

Syntax:	__PCH__
Elements:	None
Purpose:	The PCH compiler defines this pre-processor identifier. It may be used to determine if the PCH compiler is doing the compilation.

Examples:	`#ifdef _ _ PCH _ _`
	`#device PIC18C452`
	`#endif`

Example Files:	ex_sqw.c
Also See:	__PCB__, __PCM__

#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 Files:	ex_cust.c
Also See:	None

#PRIORITY

Syntax:	#priority ***ints***
Elements:	***ints*** is a list of one or more interrupts separated by commas. ***export*** makes the functions generated from this directive available to other compilation units within the link.
Purpose:	The priority directive may be used to set the interrupt priority. The highest priority items are first in the list. If an interrupt is active it is never interrupted. If two interrupts occur at around the same time then the higher one in this list will be serviced first. When linking multiple compilation units be aware only the one in the last compilation unit is used.
Examples:	`#priority rtcc,rb`
Example Files:	None
Also See:	#int_xxxx

#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 PIC16C74` `#RESERVE  0x60:0X6f`
Example Files:	ex_cust.c
Also See:	#org

#ROM

Syntax:	#rom ***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. When linking multiple compilation units be aware this directive applies to the final object file.
Examples:	#rom 0x2100={1,2,3,4,5,6,7,8}
Example Files:	None
Also See:	#org

#SEPARATE

Syntax:	#separate
Elements:	None
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`
	`swapbyte (int *a, int *b) {`
	`int t;`
	`  t=*a;`
	`  *a=*b;`
	`  *b=t;`
	`}`

Example Files:	ex_cust.c
Also See:	#inline

#SERIALIZE

Syntax: #serialize(***id=xxx***, ***next="x"*** | ***file="filename.txt"*** " | ***listfile="filename.txt"***, ***"prompt="text"***, ***log="filename.txt"***) -

Or-#serialize(***dataee=x***, ***binary=x***, ***next="x"*** | ***file="filename.txt"*** | ***listfile="filename.txt"***, ***prompt="text"***, ***log="filename.txt"***)

Elements: ***id=xxx*** Specify a C CONST identifier, may be int8, int16, int32 or char array

Use in place of id parameter, when storing serial number to EEPROM:
dataee=x The address x is the start address in the data EEPROM.
binary=x The integer x is the number of bytes to be written to address specified.
-or-
string=x The integer x is the number of bytes to be written to address specified.

Use only one of the next three options:
file="filename.txt" The file x is used to read the initial serial number from, and this file is updated by the ICD programmer. It is assumed this is a one line file with the serial number. The programmer will increment the serial number.

listfile="filename.txt" The file x is used to read the initial serial number from, and this file is updated by the ICD programmer. It is assumed this is a file one serial number per line. The programmer will read the first line then delete that line from the file. next="x" The serial number X is used for the first load, then the hex file is updated to increment x by one.

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

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

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

Examples:

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

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

//Retrieves serial number from serials.txt
#serialize(id=serialNumA,listfile="serials.txt")

//Place serial number at EEPROM address 0, reserving 1 byte
#serialize(dataee=0,binary=1,next="45",prompt="Put in Serial
number")

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

Example Files: None

Also See: None

#TASK

The RTOS is only included with the PCW and PWH 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.

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

The rate option is used to specify how often the task should execute. This must be a multiple of the minor_cycle option if one is specified in the #use rtos directive.

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

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

Examples:

```
#task(rate=1s, max=20ms, queue=5)
```

Also See: #use rtos

__ TIME __

Syntax:	__TIME__
Elements:	None
Purpose:	This pre-processor identifier is replaced at compile time with the time of the compile in the form: "hh:mm:ss"
Examples:	`printf("Software was compiled on ");` `printf(__TIME__);`
Example Files:	None
Also See:	None

#TYPE

Syntax:	#type ***standard-type*=*size*** #type ***default=area*** #type **unsigned** #type **signed**
Elements:	***standard-type*** is one of the C keywords short, int, long, or default ***size*** is 1,8,16 or 32 ***area*** is a memory region defined before the #TYPE using the addressmod directive
Purpose:	By default the compiler treats SHORT as one bit, INT as 8 bits, and LONG as 16 bits. The traditional C convention is to have INT defined as the most efficient size for the target processor. This is why it is 8 bits on the PIC®. 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. Since #TYPE may render some sizes inaccessible (like a one bit int in the above) four keywords representing the four ints may always be used: INT1, INT8, INT16, and INT32. Be warned CCS example programs and include files may not work right if you use #TYPE in your program. 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.

Examples:	`#TYPE SHORT=8, INT=16, LONG=32`
	
	`#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`
	`...`
	`void main()`
	`{`
	`int variable1; // variable1 can only take values from -128 to 127`
	`...`
	`...`
	`}`

Example Files:	ex_cust.c
Also See:	None

#UNDEF

Syntax:	#undef ***id***
Elements:	***id*** is a pre-processor id defined via #define
Purpose:	The specified pre-processor ID will no longer have meaning to the pre-processor.

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

Example Files:	None
Also See:	#define

#USE DELAY

Syntax:	#use delay (***clock***=***speed***) #use delay(***clock***=***speed***, ***restart_wdt***) #use delay(***clock=speed, type***) #use delay(***clock=speed, type=speed***) #use delay(***type=speed***)
Elements:	***speed*** is a constant 1-100000000 (1 hz to 100 mhz). This number can contain commas. It also supports the following denominations: M, MHZ, K, KHZ ***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. ***restart_wdt*** will restart the watchdog timer on every delay_us() and delay_ms() use.
Purpose:	Tells the compiler the speed of the processor and enables the use of the built-in functions: delay_ms() and delay_us(). Will also set the proper configuration bits, and if needed configure the internal oscillator. Speed is in cycles per second. An optional restart_WDT may be used to cause the compiler to restart the WDT while delaying. When linking multiple compilation units, this directive must appear in any unit that needs timing configured (delay_ms(), delay_us(), UART, SPI). In multiple clock speed applications, this directive may be used more than once. Any timing routines (delay_ms(), delay_us, UART, SPI) that need timing information will use the last defined #use delay(). For initialization purposes, the compiler will initialize the configuration bits and internal oscillator based upon the first #use delay().

Examples:	`//set timing config to 32KHz, restart watchdog timer`
	`//on delay_us() and delay_ms()`
	`#use delay(clock=32000, RESTART_WDT)`
	`//the following 4 examples all configure the timing library`
	`//to use a 20Mhz clock, where the source is an oscillator.`
	`//user must manually set HS config bit`
	`#use delay(clock=20000000)`
	`#use delay(clock=20,000,000)`
	`#use delay(clock=20M)`
	`//compiler will set HS config bit`
	`#use delay(clock=20M, oscillator)`

```
#use delay(oscillator=20M)//compiler will set HS config bit

//application is using a 10Mhz oscillator, but using the 4x
//PLL to upscale it to 40Mhz. Compiler will set H4 config bit.
#use delay(clock=40M, oscillator=10M)

//application will use the internal oscillator at 8MHz.
//compiler will set INTOSC_IO config bit, and set the internal
//oscillator to 8MHz.
#use delay(internal=8M)
)
```

Example Files:	ex_sqw.c
Also See:	delay_ms(), delay_us()

#USE FAST_IO

Syntax:	#use fast_io (***port***)
Elements:	***port*** is A-G
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 user must ensure the direction register is set correctly via set_tris_X(). When linking multiple compilation units be aware this directive only applies to the current compilation unit.
Examples:	`#use fast_io(A)`
Example Files:	ex_cust.c
Also See:	#use fixed_io, #use standard_io, set_tris_X()

#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 Files:	None
Also See:	#use fast_io, #use standard_io

#USE I2C

Syntax: #use i²c (***options***)

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

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. Specifies the speed.
SLOW	Use the slow I²C specification
RESTART_WDT	Restart the WDT while waiting in I²C_READ
FORCE_HW	Use hardware 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.

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

Examples:

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

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

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

Example Files: ex_extee.c with 16c74.h

Also See: i²c_read(), i²c_write()

#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 though 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 work both UART and non-UART ports.
	TIMEOUT=X	To set the time getc() waits for a byte in milliseconds. If no character comes in within this time the RS232_ERRORS is set to 0 as well as the return value form getc(). This works for both UART and non-UART ports.
	SYNC_SLAVE	Makes the RS232 line a synchronous slave, making the receive pin a clock in, and the data pin the data in/out.
	SYNC_MASTER	Makes the RS232 line a synchronous master, making the receive pin a clock out, and the data pin the data in/out.
	SYNC_MATER_CONT	Makes the RS232 line a synchronous master mode in continuous receive mode. The receive pin is set as a clock out, and the data pin is set as the data in/out.
	UART1	Sets the XMIT= and RCV= to the chips first hardware UART.
	UART2	Sets the XMIT= and RCV= to the chips second hardware UART.

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 SCI and the SCI 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 Files:	ex_cust.c
Also See:	getc(), putc(), printf(), setup_uart( ), RS232 I/O overview

#USE RTOS

The RTOS is only included with the PCW and PCWH packages. The CCS Real Time Operating System (RTOS) allows a PIC micro controller to run regularly scheduled tasks without the need for interrupts. This is accomplished by a function (RTOS_RUN()) that acts as a dispatcher. When a task is scheduled to run, the dispatch function gives control of the processor to that task. When the task is done executing or does not need the processor anymore, control of the processor is returned to the dispatch function which then will give control of the processor to the next task that is scheduled to execute at the appropriate time. This process is called cooperative multi-tasking.

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

Elements: options are separated by comma and may be:

timer=X	Where x is 0-4 specifying the timer used by the RTOS.
minor_cycle=time	Where time is a number followed by s, ms, us, ns. This is the longest time any task will run. Each task's execution rate must be a multiple of this time. The compiler can calculate this if it is not specified.
statistics	Maintain min, max, and total time used by each task.

Purpose: This directive tells the compiler which timer on the PIC to use for monitoring and when to grant control to a task. Changes to the specified timer's prescaler will effect the rate at which tasks are executed.

This directive can also be used to specify the longest time that a task will ever take to execute with the minor_cycle option. This simply forces all task execution rates to be a multiple of the minor_cycle before the project will compile successfully. If the this option is not specified the compiler will use a minor_cycle value that is the smallest possible factor of the execution rates of the RTOS tasks.

If the statistics option is specified then the compiler will keep track of the minimum processor time taken by one execution of each task, the maximum processor time taken by one execution of each task, and the total processor time used by each task.

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

Examples:

```
#use rtos(timer=0, minor_cycle=20ms)
```

Also See: #task

#USE SPI

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

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

MASTER	Set the device as the master.
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 of BAUD= is used).
CLOCK_LOW=n	Low time of clock in us (not needed of BAUD= is used).
DI=pin	Optional pin for incoming data.
DO=pin	Optional pin for outgoing data.
CLK=pin	Clock pin.
MODE=n	The mode to put the SPI bus.
ENABLE=pin	Optional pin to be active during data transfer.
LOAD=pin	Optional pin to be pulsed active after data is transferred.
DIAGNOSTIC=pin	Optional pin to the set high when data is sampled.
SAMPLE_RISE	Sample on rising edge.
SAMPLE_FALL	Sample on falling edge (default).
BITS=n	Max number of bits in a transfer.
SAMPLE_COUNT=n	Number of samples to take (uses majority vote).
LOAD_ACTIVE=n	Active state for LOAD pin (0, 1).
ENABLE_ACTIVE=n	Active state for ENABLE pin (0, 1).
IDLE=n	Inactive state for CLK pin (0, 1).
ENABLE_DELAY=n	Time in us to delay after ENABLE is activated.
LSB_FIRST	LSB is sent first.
MSB_FIRST	MSB is sent first.
STREAM=id	Specify a stream name for this protocol.
FORCE_HW	Forces hardware for this stream.

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 FORCE_HW option 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.

Examples:	`#use spi(DI=PIN_B1, DO=PIN_B0, CLK=PIN_B2, ENABLE=PIN__B4`
	`// uses software SPI`
	
	`#use spi(FORCE_HW, BITS=16, stream=SPI_STREAM)`
	`// uses hardware SPI and gives this stream the name`
	`// SPI_STREAM`

Example Files:	none
Also See:	spi_xfer()

#USE STANDARD_IO

Syntax:	#USE STANDARD_IO (***port***)
Elements:	***port*** may be A-G
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 Files:	ex_cust.c
Also See:	#use fast_io, #use fixed_io

#ZERO_RAM

Syntax:	#zero_ram
Elements:	None
Purpose:	This directive zero's out all of the internal registers that may be used to hold variables before program execution begins.

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

Example Files:	ex_cust.c
Also See:	None

BUILT-IN-FUNCTIONS

BUILT-IN-FUNCTIONS

The CCS compiler provides a lot of built-in functions to access and use the PIC microcontroller 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. A complete description, parameter and return value descriptions follow in the subsequent pages.

RS232 I/O	ASSERT( )	GETCH( )	PUTC( )	
	FGETC( )	GETCHAR( )	PUTCHAR( )	
	FGETS( )	GETS( )	PUTS( )	
	FPRINTF( )	KBHIT( )	SET_UART_SPEED( )	
	FPUTC( )	PERROR( )	SETUP_UART( )	
	FPUTS( )	PRINTF( )		
SPI TWO WIRE I/O	SETUP_SPI( )	SPI_DATA_IS_IN( )	SPI_READ( )	SPI_WRITE( )
	SPI_XFER( )			
DISCRETE I/O	GET_TRISx()	INPUT_K( )	OUTPUT_FLOAT()	SET_TRIS_B( )
	INPUT( )	INPUT_STATE( )	OUTPUT_G( )	SET_TRIS_C( )
	INPUT_A( )	INPUT_x( )	OUTPUT_H( )	SET_TRIS_D( )
	INPUT_B( )	OUTPUT_A( )	OUTPUT_HIGH( )	SET_TRIS_E( )
	INPUT_C( )	OUTPUT_B( )	OUTPUT_J( )	SET_TRIS_F( )
	INPUT_D( )	OUTPUT_BIT( )	OUTPUT_K( )	SET_TRIS_G( )
	INPUT_E( )	OUTPUT_C( )	OUTPUT_LOW( )	SET_TRIS_H( )
	INPUT_F( )	OUTPUT_D( )	OUTPUT_TOGGLE ( )	SET_TRIS_J( )
	INPUT_G( )	OUTPUT_DRIVE( )	PORT_A_PULLUP S()	SET_TRIS_K( )
	INPUT_H( )	OUTPUT_E( )	PORT_B_PULLUP S()	
	INPUT_J( )	OUTPUT_F( )	SET_TRIS_A( )	
PARALLEL SLAVE I/O	PSP_INPUT_FULL( )		PSP_OVERFLOW( )	

	PSP_OUTPUT_FULL( )	SETUP_PSP( )		
I²C I/O	I²C_ISR_STATE()	I²C_READ( )	I²C_STOP( )	
	I²C_POLL( )	I²C_START( )	I²C_WRITE( )	
PROCESSOR CONTROLS	CLEAR_INTERRUPT( )	GOTO_ADDRESS( )	RESET_CPU( )	
	DISABLE_INTERRUPTS( )	INTERRUPT_ACTIVE( )	RESTART_CAUSE( )	
	ENABLE_INTERRUPTS( )	JUMP_TO_ISR	SETUP_OSCILLATOR()	
	EXT_INT_EDGE( )	LABEL_ADDRESS( )	SLEEP( )	
	GETENV( )	READ_BANK( )	WRITE_BANK( )	
BIT/BYTE MANIPULATION	BIT_CLEAR( )	MAKE8( )	_MUL( )	SHIFT_LEFT( )
	BIT_SET( )	MAKE16( )	ROTATE_LEFT( )	SHIFT_RIGHT()
	BIT_TEST( )	MAKE32( )	ROTATE_RIGHT()	SWAP( )
STANDARD C MATH	ABS( )	COSH( )	LABS( )	SIN( )
	ACOS( )	DIV( )	LDEXP( )	SINH( )
	ASIN( )	EXP( )	LDIV( )	SQRT( )
	ATAN( )	FABS( )	LOG( )	TAN( )
	ATAN2( )	FLOOR( )	LOG10( )	TANH( )
	CEIL( )	FMOD( )	MODF( )	
	COS( )	FREXP( )	POW( )	
VOLTAGE REF	SETUP_LOW_VOLT_DETECT( )	SETUP_VREF( )		
A/D CONVERSION	SET_ADC_CHANNEL( )	SETUP_ADC_PORTS( )		
	SETUP_ADC( )	READ_ADC( )		
STANDARD C CHAR	ATOF( )	ISLOWER(char)	STRCMP( )	STRRCHR( )
	ATOI( )	ISPRINT(x)	STRCOLL( )	STRSPN( )
	ATOI32( )	ISPUNCT(x)	STRCPY( )	STRSTR( )
	ATOL( )	ISSPACE(char)	STRCSPN( )	STRTOD( )

	ISALNUM( )	ISUPPER(char)	STRLEN( )	STRTOK( )
	ISALPHA(char)	ISXDIGIT(char)	STRLWR( )	STRTOL( )
	ISAMOUNG( )	ITOA( )	STRNCAT( )	STRTOUL( )
	ISCNTRL(x)	SPRINTF( )	STRNCMP( )	STRXFRM( )
	ISDIGIT(char)	STRCAT( )	STRNCPY( )	TOLOWER( )
	ISGRAPH(x)	STRCHR( )	STRPBRK( )	TOUPPER( )

TIMERS	GET_TIMER0( )	SET_RTCC( )	SETUP_TIMER_0 ( )
	GET_TIMER1( )	SET_TIMER0( )	SETUP_TIMER_1 ( )
	GET_TIMER2( )	SET_TIMER1( )	SETUP_TIMER_2 ( )
	GET_TIMER3( )	SET_TIMER2( )	SETUP_TIMER_3 ( )
	GET_TIMER4( )	SET_TIMER3( )	SETUP_TIMER_4 ( )
	GET_TIMER5( )	SET_TIMER4( )	SETUP_TIMER_5 ( )
	GET_TIMERx( )	SET_TIMER5( )	SETUP_WDT ( )
	RESTART_WDT( )	SETUP_COUNTERS( )	

STANDARD C MEMORY	CALLOC( )	MEMCMP( )	OFFSETOFBIT( )
	FREE( )	MEMCPY( )	REALLOC( )
	LONGJMP( )	MEMMOVE( )	SETJMP( )
	MALLOC( )	MEMSET( )	
	MEMCHR( )	OFFSETOF( )	

CAPTURE/COMPARE/PWM	SET_POWER_PWM_OVERRIDE( )	SETUP_CCP2( )
	SET_POWER_PWMX_DUTY( )	SETUP_CCP3( )
	SET_PWM1_DUTY( )	SETUP_CCP4( )
	SET_PWM2_DUTY( )	SETUP_CCP5( )
	SET_PWM3_DUTY( )	SETUP_CCP6( )
	SET_PWM4_DUTY( )	SETUP_POWER_PWM( )
	SET_PWM5_DUTY( )	SETUP_POWER_PWM_PINS( )
	SETUP_CCP1( )	

INTERNAL EEPROM	ERASE_PROGRAM_EEPROM( )	SETUP_EXTERNAL_MEMORY( )
	READ_CALIBRATION( )	WRITE_CONFIGURATION_MEMORY()
	READ_EEPROM( )	WRITE_EEPROM( )
	READ_EXTERNAL_MEMORY( )	WRITE_EXTERNAL_MEMORY( )
	READ_PROGRAM_EEPROM( )	WRITE_PROGRAM_EEPROM( )
	READ_PROGRAM_MEMORY( )	WRITE_PROGRAM_MEMORY( )

STANDARD C SPECIAL	BSEARCH( )	RAND( )	SRAND( )	QSORT( )

DELAYS	DELAY_CYCLES( )	DELAY_MS( )	DELAY_US( )
ANALOG COMPARE	SETUP_COMPARATOR( )		
RTOS	RTOS_AWAIT( ) RTOS_DISABLE( ) RTOS_ENABLE( ) RTOS_MSG_POLL( ) RTOS_MSG_READ( )	RTOS_MSG_SEND( ) RTOS_OVERRUN( ) RTOS_RUN( ) RTOS_SIGNAL( ) RTOS_STATS( )	RTOS_TERMINATE( ) RTOS_WAIT( ) RTOS_YIELD( )
STANDARD STRING	STRXFRM( ) STRCAT( ) STRCOLL( ) STRCOLL( ) STRLEN( ) STRNCMP( ) STRRCHR( ) STANDARD STRING FUNCTION( )	MEMCHR( ) STRCHR( ) STRCSPN( ) STRCSPN( ) STRLWR( ) STRNCPY( ) STRSPN( )	MEMCMP( ) STRCMP( ) STRICMP( ) STRICMP( ) STRNCAT( ) STRPBRK( ) STRSTR( )
LCD	LCD_LOAD( )	LCD_SYMBOL( )	SETUP_LCD( )
MISC.	SETUP_OPAMP1( )	SETUP_OPAMP2( )	SLEEP_ULPWU( )

ABS( )

Syntax:	value = abs(***x***)
Parameters:	***x*** is a signed 8, 16, or 32 bit int or a float
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()

ACOS( )

See: SIN()

ASIN( )

See: SIN()

ASSERT( )

Syntax:	assert (***condition***);
Parameters:	***condition*** is any relational expression
Returns:	Nothing
Function:	This function tests the condition and if FALSE will generate an error message on STDERR (by default the first USE RS232 in the program). The error message will include the file and line of the assert(). No code is generated for the assert() if you #define NODEBUG. In this way you may include asserts in your code for testing and quickly eliminate them from the final program.
Availability:	All devices
Requires:	assert.h and #use rs232

Examples:	`assert( number_of_entries<TABLE_SIZE );`
	
	`// If number_of_entries is >= TABLE_SIZE then`
	`// the following is output at the RS232:`
	`// Assertion failed, file myfile.c, line 56`

Example Files:	None
Also See:	#use rs232, RS232 I/O overview

ATAN( )

See: SIN()

ATAN2( )

See: SIN()

ATOF( )

Syntax:	result = atof (***string***)
Parameters:	***string*** is a pointer to a null terminated string of characters.
Returns:	Result is a 32 bit 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.
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()

ATOI( ), ATOL( ), ATOI32( )

Syntax:	ivalue = atoi(***string***) or lvalue = atol(***string***) or i32value = atoi32(***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.
Function:	Converts the string pointed to by ptr to 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()

ATOI32( )
ATOL( )

See ATOI( )

BIT_CLEAR( )

Syntax:	bit_clear(***var***, ***bit***)
Parameters:	***var*** may be a 8,16 or 32 bit variable (any lvalue) ***bit*** is a number 0-31 representing a bit number, 0 is the least significant bit.
Returns:	undefined
Function:	Simply clears the specified bit (0-7, 0-15 or 0-31) in the given variable. The least significant bit is 0. This function is the same as: var &= ~(1<<bit);
Availability:	All devices
Requires:	Nothing

Examples:	`int x;`
	`x=5;`
	`bit_clear(x,2);`
	`// x is now 1`
	
	`bit_clear(*11,7); // A crude way to disable ints`

Example Files:	ex_patg.c
Also See:	bit_set(), bit_test()

BIT_SET( )

Syntax:	bit_set(***var***, ***bit***)
Parameters:	var may be a 8,16 or 32 bit variable (any lvalue) bit is a number 0-31 representing a bit number, 0 is the least significant bit.
Returns:	Undefined
Function:	Sets the specified bit (0-7, 0-15 or 0-31) in the given variable. The least significant bit is 0. This function is the same as: var |= (1<<bit);
Availability:	All devices
Requires:	Nothing

Examples:	`int x;`
	`x=5;`
	`bit_set(x,3);`
	`// x is now 13`
	
	`bit_set(*6,1); // A crude way to set pin B1 high`

Example Files:	ex_patg.c
Also See:	bit_clear(), bit_test()

BIT_TEST( )

Syntax:	value = bit_test (***var***, ***bit***)
Parameters:	***var*** may be a 8,16 or 32 bit variable (any lvalue) ***bit*** is a number 0-31 representing a bit number, 0 is the least significant bit.
Returns:	0 or 1
Function:	Tests the specified bit (0-7,0-15 or 0-31) in the given variable. The least significant bit is 0. This function is much more efficient than, but otherwise the same as: ((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_patg.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 and size the number of bytes to be allocated or 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:	STDLIBM.H must be included

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 a float
Returns:	A float
Function:	Computes the smallest integral value greater than the argument. CEIL(12.67) is 13.00.
Availability:	All devices
Requires:	#include <math.h>

Examples:	`// Calculate cost based on weight rounded`
	`// up to the next pound`
	
	`cost = ceil( weight ) * DollarsPerPound;`

Example Files:	None
Also See:	floor()

CLEAR_INTERRUPT( )

Syntax:	clear_interrupt(***level***)
Parameters:	level - a constant defined in the devices.h file
Returns:	undefined
Function:	Clears the interrupt flag for the given level. This function is designed for use with a specific interrupt, thus eliminating the GLOBAL level as a possible parameter. Some chips that have interrupt on change for individual pins allow the pin to be specified like INT_RA1.
Availability:	All devices
Requires:	Nothing
Examples:	`clear_interrupt(int_timer1);`
Example Files:	None
Also See:	enable_interrupts(), #INT, Interrupts overview

COS( )

See: SIN()

COSH( )

See: SIN()

DELAY_CYCLES( )

Syntax:	delay_cycles (***count***)
Parameters:	***count*** - a constant 1-255
Returns:	undefined
Function:	Creates code to perform a delay of the specified number of instruction clocks (1-255). An instruction clock is equal to four oscillator clocks. The delay time may be longer than requested if an interrupt is serviced during the delay. The time spent in the ISR does not count toward the delay time.
Availability:	All devices
Requires:	Nothing

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

Example Files:	ex_cust.c
Also See:	delay_us(), delay_ms()

DELAY_MS( )

Syntax:	delay_ms (***time***)
Parameters:	***time*** - a variable 0-65535(int16) or a constant 0-65535 Note: Previous compiler versions ignored the upper byte of an int16, now the upper byte affects the time.
Returns:	undefined
Function:	This function will create code to perform a delay of the specified length. Time is specified in milliseconds. This function works by executing a precise number of instructions to cause the requested delay. It does not use any timers. If interrupts are enabled the time spent in an interrupt routine is not counted toward the time. The delay time may be longer than requested if an interrupt is serviced during the delay. The time spent in the ISR does not count toward the delay time.
Availability:	All devices
Requires:	#use delay

Examples:	`#use delay (clock=20000000)`
	
	`delay_ms( 2 );`
	
	`void delay_seconds(int n) {`
	`  for (;n!=0; n- -)`
	`  delay_ms( 1000 );`
	`}`

Example Files:	ex_sqw.c
Also See:	delay_us(), delay_cycles(), #use delay

DELAY_US( )

Syntax:	delay_us (***time***)
Parameters:	***time*** - a variable 0-65535(int16) or a constant 0-65535 Note: Previous compiler versions ignored the upper byte of an int16, now the upper byte affects the time.
Returns:	undefined
Function:	Creates code to perform a delay of the specified length. Time is specified in microseconds. Shorter delays will be INLINE code and longer delays and variable delays are calls to a function. This function works by executing a precise number of instructions to cause the requested delay. It does not use any timers. If interrupts are enabled the time spent in an interrupt routine is not counted toward the time. The delay time may be longer than requested if an interrupt is serviced during the delay. The time spent in the ISR does not count toward the delay time.
Availability:	All devices
Requires:	#use delay

Examples:	`#use delay(clock=20000000)`
	
	`do {`
	`output_high(PIN_B0);`
	`delay_us(duty);`
	`output_low(PIN_B0);`
	`delay_us(period-duty);`
	`} while(TRUE);`

Example Files:	ex_sqw.c
Also See:	delay_ms(), delay_cycles(), #use delay

DISABLE_INTERRUPTS( )

Syntax:	disable_interrupts (***level***)
Parameters:	***level*** - a constant defined in the devices .h file
Returns:	undefined
Function:	Disables the interrupt at the given level. The GLOBAL level will not disable any of the specific interrupts but will prevent any of the specific interrupts, previously enabled to be active. Valid specific levels are the same as are used in #INT_xxx and are listed in the devices .h file. GLOBAL will also disable the peripheral interrupts on devices that have it. Note that it is not necessary to disable interrupts inside an interrupt service routine since interrupts are automatically disabled. Some chips that have interrupt on change for individual pins allow the pin to be specified like INT_RA1.
Availability:	Device with interrupts (PCM and PCH)
Requires:	Should have a #int_xxxx, constants are defined in the devices .h file.

Examples:	`disable_interrupts(GLOBAL); // all interrupts OFF`
	`disable_interrupts(INT_RDA); // RS232 OFF`
	
	`enable_interrupts(ADC_DONE);`
	`enable_interrupts(RB_CHANGE);`
	`  // these enable the interrupts`
	`  // but since the GLOBAL is disabled they`
	`  // are not activated until the following`
	`  // statement:`
	`enable_interrupts(GLOBAL);`

Example Files:	None
Also See:	enable_interrupts(), #int_xxxx, Interrupts overview

DIV( ), LDIV( )

Syntax:	idiv=div(***num***, ***denom***) ldiv =ldiv(***lnum***, ***ldenom***) idiv=ldiv(***lnum***, ***ldenom***)
Parameters:	***num*** and ***denom*** are signed integers. ***num*** is the numerator and ***denom*** is the denominator. ***lnum*** and ***ldenom*** are signed longs. ***lnum*** is the numerator and ***ldenom*** is the denominator.
Returns:	idiv is an object of type div_t and lidiv is an object of type ldiv_t. The div function returns a structure of type div_t, comprising of both the quotient and the remainder. The ldiv function returns a structure of type ldiv_t, comprising of both the quotient and the remainder.
Function:	The div and ldiv function computes the quotient and remainder of the division of the numerator by the denominator. If the division is inexact, the resulting quotient is the integer or long of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise quot*denom(ldenom)+rem shall equal num(lnum).
Availability:	All devices.
Requires:	#include <STDLIB.H>

Examples:	`div_t idiv;`
	`ldiv_t lidiv;`
	`idiv=div(3,2);`
	`// idiv will contain quot=1 and rem=1`
	
	`lidiv=ldiv(300,250);`
	`//lidiv will contain quot=1 and rem=50`

Example Files:	None
Also See:	None

ENABLE_INTERRUPTS( )

Syntax:	enable_interrupts (***level***)
Parameters:	***level*** - a constant defined in the devices .h file
Returns:	undefined
Function:	Enables the interrupt at the given level. An interrupt procedure should have been defined for the indicated interrupt. The GLOBAL level will not enable any of the specific interrupts but will allow any of the specific interrupts previously enabled to become active. Some chips that have interrupt on change for individual pins allow the pin to be specified like INT_RA1.
Availability:	Device with interrupts (PCM and PCH)
Requires:	Should have a #int_xxxx, Constants are defined in the devices .h file.

Examples:	`enable_interrupts(GLOBAL);`
	`enable_interrupts(INT_TIMER0);`
	`enable_interrupts(INT_TIMER1);`

Example Files:	None
Also See:	disable_enterrupts(), #int_xxxx, Interrupts overview

ERASE_PROGRAM_EEPROM( )

Syntax:	erase_program_eeprom (***address***);
Parameters:	***address*** is 16 bits on PCM parts and 32 bits on PCH parts. 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. For example, if it is 64 bytes then the least significant 6 bits of address is ignored. See WRITE_PROGRAM_MEMORY 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+=getenv("FLASH_ERASE_SIZE"))` `erase_program_memory(i);`
Example Files:	None
Also See:	WRITE_PROGRAM_EEPROM(), WRITE_PROGRAM_MEMORY(), Program eeprom overview

EXP( )

Syntax:	result = exp (***value***)
Parameters:	***value*** is a float
Returns:	A float
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:	math.h must be included

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

Example Files:	None
Also See:	pow(), log(), log10()

EXT_INT_EDGE( )

Syntax:	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_wakup.c
Also See:	#INT_EXT, enable_interrupts(), disable_interrupts(), Interrupts overview

FABS( )

Syntax:	result=fabs (***value***)
Parameters:	***value*** is a float
Returns:	result is a float
Function:	The fabs function computes the absolute value of a float
Availability:	All devices.
Requires:	MATH.H must be included

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

Example Files:	None
Also See:	abs(), labs()

FGETC( )

See GETC

FGETS( )

See GETS

FLOOR( )

Syntax:	result = floor (***value***)
Parameters:	***value*** is a float
Returns:	result is a float
Function:	Computes the greatest integral value not greater than the argument. Floor (12.67) is 12.00.
Availability:	All devices.
Requires:	MATH.H must be included

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*** and ***val2*** are floats
Returns:	result is a float
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:	MATH.H must be included

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

Example Files:	None
Also See:	None

FPRINTF( )

See PRINTF

FPUTC( )

See PUTC

FPUTS( )

See PUTS

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:	STDLIBM.H must be included

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

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

FREXP( )

Syntax:	result=frexp (***value***, & ***exp***);
Parameters:	***value*** is float ***exp*** is a signed int.
Returns:	result is a float
Function:	The frexp function breaks a floating point number into a normalized fraction and an integral power of 2. It stores the integer in the signed int object exp. The result is in the interval [1/2,1) 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:	MATH.H must be included

Examples:	`float result;`
	`signed int exp;`
	`result=frexp(.5,&exp);`
	`//  result is .5 and exp is 0`

Example Files:	None
Also See:	ldexp(), exp(), log(), log10(), modf()

GET_TIMERx( )

Syntax:	value=get_timer0() Same as: value=get_rtcc() value=get_timer1() value=get_timer2() value=get_timer3() value=get_timer4() value=get_timer5()
Parameters:	None
Returns:	Timers 1, 3, and 5 return a 16 bit int. Timers 2 and 4 return an 8 bit int. Timer 0 (AKA RTCC) returns a 8 bit int except on the PIC18XXX where it returns a 16 bit int.
Function:	Returns 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 Timer 4 - Some PCH devices Timer 5 - Only PIC18XX31
Requires:	Nothing
Examples:	`set_timer0(0);` `while ( get_timer0() < 200 ) ;`
Example Files:	ex_stwt.c
Also See:	set_timerx(), setup_timerx(), Timer0 overview, Timer1 overview, Timer2 overview, Timer5 overview

GET_TRISx( )

Syntax:	value = get_tris_A();
	value = get_tris_B();
	value = get_tris_C();
	value = get_tris_D();

Parameters:	None
Returns:	Byte, the value of TRIS register
Function:	Returns the value of the TRIS register of port A , B , C or D.
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

GETCHAR( )

See GETC

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 Returns 1 if fuse fffff is valid
INT:iiiii	Returns 1 if the interrupt iiiii is valid
ID	Returns the device ID (set by #ID)
DEVICE	Returns the device name string (like "PIC16C74")
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
DATA_EEPROM	Returns the number of bytes of data EEPROM
READ_PROGRAM	Returns a 1 if the code memory can be read
PIN:pb	Returns a 1 if bit b on port p is on this part
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
	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 #byte. ***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)

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”)
#bit carry_flag=GETENV(“BIT:C”)
```

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 Files:	None
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 Files:	setjmp.h
Also See:	label_address( )

I2C_ISR_STATE( )

Syntax:	state = i²c_isr_state();
Parameters:	None
Returns:	state is an 8 bit int 0 - Address match received with R/W bit clear 1-0x7F - Master has written data; i²c_read() will immediately return the data 0x80 - Address match received with R/W bit set; respond with i²c_write() 0x81-0xFF - Transmission completed and acknowledged; respond with i²c_write()
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.
Availability:	Devices with i²c hardware
Requires:	#use i²c

Examples:	`#INT_SSP`
	`  void i2c_isr() {`
	`  state = i2c_isr_state();`
	`  if(state >= 0x80)`
	`  i2c_write(send_buffer[state - 0x80]);`
	`  else if(state > 0)`
	`  rcv_buffer[state - 1] = i2c_read();`
	`}`

Example Files:	None
Also See:	i²c_write, i²c_read, #usei²c

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:	`i2c_start();     // Start condition`
	`i2c_write(0xc1); // Device address/Read`
	`count=0;`
	`while(count!=4) {`
	`  while(!i2c_poll()) ;`
	`  buffer[count++]= i2c_read();  //Read Next`
	`}`
	`i2c_stop();      // Stop condition`

Example Files:	ex_slave.c
Also See:	i²c_start, i²c_write, i²c_stop ,I²C overview

I2C_READ( )

Syntax:	data = i²c_read(); data = i²c_read(stream); data = i²c_read(stream, ack);
Parameters:	***ack*** -Optional, defaults to 1. 0 indicates do not ack. 1 indicates to ack. **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:	Devices with built in I²C
Requires:	#use i²c

Examples:	`i2c_start();`
	`i2c_write(0xa1);`
	`data1 = i2c_read();`
	`data2 = i2c_read();`
	`i2c_stop();`

Example Files:	ex_extee.c with 2416.c
Also See:	i²c_start, i²c_write, i²c_stop, i²c_poll, I²C overview

I2C_SlaveAddr( )

Syntax:	I2C_SlaveAdr(int8 addr); I2C_SlaveAddr(stream, int8 addr);
Parameters:	Addr = 8b long deveice address Stream = specifies which I2C stream declared in #use I2C statement is used (optional)
Returns:	Nothing
Function:	This function 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_start, i²c_write, i²c_stop, i²c_poll, 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 uncounted 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 stopp'ed 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_write, i²c_stop, i²c_poll, 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_start, i²c_write, i²c_read, i²c_poll, #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.
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_start(), i²c_stop, i²c_read, i²c_poll, #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 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43. The PIN could also be a variable. 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_I0 mode is set on port A. note that doing I/0 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");`
	
	`int16i=PIN_B1;`
	`while(!i);`
	`//waits for B1 to go high`

Example Files:	ex_pulse.c
Also See:	input_x(), output_low(), output_high(), #use xxxx_io

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 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43.
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 state of a pin without changing the direction of the pin as INPUT() does.
Availability:	All devices.
Requires:	Nothing
Examples:	`level = input_state(pin_A3);` `printf("level: %d",level);`
Example Files:	None
Also See:	input(), set_tris_x(), output_low(), output_high()

INPUT_x( )

Syntax:	value = input_a() value = input_d() value = input_g() value = input_k()	value = input_b() value = input_e() value = input_h()	value = input_c() value = input_f() value = input_j()
Parameters:	None		
Returns:	An 8 bit int representing the port input data.		
Function:	Inputs an entire byte 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:			
Also See:	input(), output_x(), #USE xxxx_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 (PCM and PCH)
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_enterrupts(), #INT, Interrupts overview

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 = isdigit(***datac***)
value = islower(***datac***)
value = isspace(***datac***)
value = isupper(***datac***)
value = isxdigit(***datac***)
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

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

Availability:

All devices.

Requires:

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:

isamoung()

ISAMOUNG( )

Syntax:	result = isamoung (***value***, ***cstring***)
Parameters:	***value*** is a character ***cstring*** is a constant string
Returns:	0 (or FALSE) if value is not in cstring 1 (or TRUE) if value is in cstring
Function:	Returns TRUE if a character is one of the characters in a constant string.
Availability:	All devices.
Requires:	Nothing

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

Example Files:	ctype.h
Also See:	isalnum(), isalpha(), isdigit(), isspace(), islower(), isupper(), isxdigit()

ITOA( )

Syntax:	**string** = itoa(**i32value**, **i8base, string**)
Parameters:	**i32value** is a 32 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 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:	#inlcude<stdlib.h>

Examples:	`int32 x=1234;`
	`char string[5];`
	
	`itoa(x,10, string);`
	// string is now “1234”

Example Files:	None
Also See:	None

JUMP_TO_ISR

Syntax:	jump_to_isr (***address***)
Parameters:	***address*** is a valid program memory address
Returns:	No value
Function:	The jump_to_isr function is used when the location of the interrupt service routines are not at the default location in program memory. When an interrupt occurs, program execution will jump to the default location and then jump to the specified address.
Availability:	All devices
Requires:	Nothing

Examples:	`int_global`
	`void global_isr(void) {`
	`  jump_to_isr(isr_address);`
	`}`

Example Files:	None
Also See:	#build( )

KBHIT( )

Syntax: value = kbhit()
value = kbhit (***stream***)

Parameters: ***stream*** is the stream id assigned to an available RS232 port. If the stream parameter is not included, the function uses the primary stream used by getc().

Returns: 0 (or FALSE) if getc() will need to wait for a character to come in, 1 (or TRUE) if a character is ready for getc()

Function: If the RS232 is under software control this function returns TRUE if the start bit of a character is being sent on the RS232 RCV pin. If the RS232 is hardware this function returns TRUE if a character has been received and is waiting in the hardware buffer for getc() to read. This function may be used to poll for data without stopping and waiting for the data to appear. Note that in the case of software RS232 this function should be called at least 10 times the bit rate to ensure incoming data is not lost.

Availability: All devices.

Requires: #use rs232

Examples:

```
char timed_getc() {

   long timeout;

   timeout_error=FALSE;
   timeout=0;
   while(!kbhit()&&(++timeout<50000)) // 1/2
                                      // second
          delay_us(10);
   if(kbhit())
          return(getc());
   else {
          timeout_error=TRUE;
          return(0);
   }
}
```

Example Files: ex_tgetc.c

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

LABEL_ADDRESS( )

Syntax:	value = label_address(***label***);
Parameters:	***label*** is a C label anywhere in the function
Returns:	A 16 bit int in PCB,PCM and a 32 bit int for PCH
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.c
Also See:	goto_address()

LABS( )

Syntax:	result = labs (***value***)
Parameters:	***value*** is a 16 bit signed long int
Returns:	A 16 bit signed long int
Function:	Computes the absolute value of a long integer.
Availability:	All devices.
Requires:	stdlib.h must be included
Examples:	`if(labs( target_value - actual_value ) > 500)` `  printf("Error is over 500 points\r\n");`
Example Files:	None
Also See:	abs()

LCD_LOAD( )

Syntax:	lcd_load (***buffer_pointer***, ***offset***, ***length***);
Parameters:	***buffer_pointer*** points to the user data to send to the LCD, ***offset*** is the offset into the LCD segment memory to write the data, ***length*** is the number of bytes to transfer.
Returns:	undefined
Function:	Will load length bytes from buffer_pointer into the 923/924 LCD segment data area beginning at offset (0-15). lcd_symbol provides an easier way to write data to the segment memory.
Availability:	This function is only available on devices with LCD drive hardware.
Requires	Constants are defined in the devices .h file.
Examples:	`lcd_load(buffer, 0, 16);`
Example Files:	ex_92lcd.c
Also See:	lcd_symbol(), setup_lcd(), Internal LCD overview

LCD_SYMBOL( )

Syntax:	lcd_symbol (symbol, bx_addr[, by_addr]);
Parameters:	***symbol*** is a 8 bit or 16 bit constant. ***bX_addr*** is a bit address representing the segment location to be used for bit X of symbol. 1-16 segments could be specified.
Returns:	undefined
Function:	Loads bits into the segment data area for the LCD with each bit address specified. If bit 0 in symbol is set the segment at B0_addr is set, otherwise it is cleared. The same is true of all other bits in symbol. The B0_addr is a bit address into the LCD RAM.
Availability:	This function is only available on devices with LCD drive hardware.
Requires	Constants are defined in the devices .h file.

Examples:	`byte CONST DIGIT_MAP[10]=`
	`{0X90,0XB7,0X19,0X36,0X54,0X50,0XB5,0X24};`
	
	`#define DIGIT_1_CONFIG`
	`COM0+2,COM0+4,COM05,COM2+4,COM2+1,`
	`COM1+4,COM1+5`
	
	`for(i=1; i<=9; ++i) {`
	`  LCD_SYMBOL(DIGIT_MAP[i],DIGIT_1_CONFIG);`
	`  delay_ms(1000);`
	`}`

Example Files:	ex_92lcd.c
Also See:	setup_lcd(), lcd_load(), Internal LCD Overview

LDEXP( )

Syntax:	result= ldexp (***value***, ***exp***);
Parameters:	***value*** is float ***exp*** is a signed int.
Returns:	result is a float with value result times 2 raised to power exp.
Function:	The ldexp function multiplies a floating-point number by an integral power of 2.
Availability:	All devices.
Requires:	MATH.H must be included

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 a float
Returns:	A float
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:	math.h must be included.
Examples:	`lnx = log(x);`
Example Files:	None
Also See:	log10(), exp(), pow()

LOG10( )

Syntax:	result = log10 (***value***)
Parameters:	***value*** is a float
Returns:	A float
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:	Itcl298.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 Files:	ex_freqc.c
Also See:	make8(), make16()

MALLOC( )

Syntax:	ptr=malloc(***size***)
Parameters:	***size*** is an integer representing the number of byes 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:	STDLIBM.H must be included

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

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

MEMCPY( ), MEMMOVE( )

Syntax:	memcpy (***destination***, ***source***, ***n***) memmove(***destination***, ***source***, ***n***)
Parameters:	***destination*** is a pointer to the destination memory, ***source*** is a pointer to the source memory, ***n*** is the number of bytes to transfer
Returns:	undefined
Function:	Copies n bytes from source to destination in RAM. Be aware that array names are pointers where other variable names and structure names are not (and therefore need a & before them). Memmove performs a safe copy (overlapping objects doesn't cause a problem). Copying takes place as if the n characters from the source are first copied into a temporary array of n characters that doesn't overlap the destination and source objects. Then the n characters from the temporary array are copied to destination.
Availability:	All devices
Requires:	Nothing

Examples:	`memcpy(&structA, &structB, sizeof (structA));`
	`memcpy(arrayA,arrayB,sizeof (arrayA));`
	`memcpy(&structA, &databyte, 1);`
	
	`char a[20]="hello";`
	`memmove(a,a+2,5);`
	`// a is now "llo"MEMMOVE()`

Example Files:	None
Also See:	strcpy(), memset()

MEMSET( )

Syntax:	memset (***destination***, ***value***, ***n***)
Parameters:	***destination*** is a pointer to memory, ***value*** is a 8 bit int, ***n*** is a 8 bit int.
Returns:	undefined
Function:	Sets value to destination for n number of bytes. 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*** and ***integral*** are floats
Returns:	result is a float
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:	MATH.H must be included

Examples:	`float 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 integers or 16-bit integers
Returns:	A 16-bit integer if both parameters are 8-bit integers, or a 32-bit integer if both parameters are 16-bit integers.
Function:	Performs an optimized multiplication. By accepting a different type than it returns, this function avoids the overhead of converting the parameters to a larger type.
Availability:	All devices
Requires:	Nothing

Examples:	`int a=50, b=100;`
	`long int c;`
	`c = _mul(a, b);  //c holds 5000`

Example Files:	None
Also See:	None

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:	stddef.h

Examples:	`struct  time_structure {`
	`  int hour, min, sec;`
	`  int zone : 4;`
	`  int1 daylight_savings;`
	`}`
	
	`x = offsetof(time_structure, sec);`
	`  // x will be 2`
	`x = offsetofbit(time_structure, sec);`
	`  // x will be 16`
	`x = offsetof (time_structure,`
	`  daylight_savings);`
	`  // x will be 3`
	`x = offsetofbit(time_structure,`
	`  daylight_savings);`
	`  // x will be 28`

Example Files:	None
Also See:	None

OFFSETOFBIT( )

See OFFSETOF

OUTPUT_A( ), OUTPUT_B( ), OUTPUT_C( ), OUTPUT_D( ), OUTPUT_E( ), OUTPUT_F( ), OUTPUT_G( ), OUTPUT_H( ), OUTPUT_J( ), OUTPUT_K( )

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 8 bit int
Returns:	undefined
Function:	Output an entire byte 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 xxxx_io

OUTPUT_B
OUTPUT_C
OUTPUT_D
OUTPUT_E
OUTPUT_F
OUTPUT_G
OUTPUT_H
OUTPUT_J
OUTPUT_K

See OUTPUT_A

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 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43. The PIN could also be a variable. 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_I0 mode is set on port A. Note that doing I/0 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`
	`int16i=PIN_B0;`
	`ouput_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.h
Also See:	input(), output_low(), output_high(), output_float(), output_x(), #use xxxx_io

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 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43.
Returns:	undefined
Function:	Sets the specified pin to the output mode. This will allow the pin to have its value read.
Availability:	All devices.
Requires:	Pin constants are defined in the devices.h file.

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

Example Files:	None
Also See:	input(), output_low(), output_high(), output_bit(), output_x(), output_float()

.

OUTPUT_FLOAT( )

Syntax:	output_float (***pin***)
Parameters:	***Pins*** are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43. The PIN could also be a variable to identify the pin. 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_I0 mode is set on port A. note that doing I/0 with a variable instead of a constant will take much longer time.
Returns:	undefined
Function:	Sets the specified pin to the input mode. This will allow the pin to float high to represent a high on an open collector type of connection.
Availability:	All devices.
Requires:	Pin constants are defined in the devices .h file

Examples:	`if( (data & 0x80)==0 )`
	`  output_low(pin_A0);`
	`else`
	`  output_float(pin_A0);`

Example Files:	None
Also See:	input(), output_low(), output_high(), output_bit(), output_x(), output_drive(), #use xxxx_io

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 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43. The PIN could also be a variable. 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_I0 mode is set on port A. Note that doing I/0 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);`
	
	`Int16i=PIN_A1;`
	`output_low(PIN_A1);`

Example Files:	None
Also See:	input(), output_low(), output_float(), output_bit(), output_x(), #use xxxx_io

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 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43. The PIN could also be a variable. 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_I0 mode is set on port A. Note that doing I/0 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:	sqw.c
Also See:	input(), output_high(), output_float(), output_bit(), output_x(), #use xxxx_io

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 5) bit 3 would have a value of 5*8+3 or 43. This is a defined as follows: #define PIN_A3 43.
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, errno.h

Examples:	`x = sin(y);`
	
	`if(errno!=0)`
	`  perror("Problem in find_area");`

Example Files:	None
Also See:	RS232 I/O overview

PORT_A_PULLUPS ( )

Syntax:	port_a_pullups (***value***)
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.
Returns:	undefined
Function:	Sets the port A 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:	ex_lcdkb.c with kdb.c
Also See:	input(), input_x(), output_float()

PORT_B_PULLUPS( )

Syntax:	port_b_pullups (***value***)
Parameters:	***value*** is TRUE or FALSE on most parts, some parts that allow pullups to be specified on individual pins permit a 8 bit int here, one bit for each port pin
Returns:	undefined
Function:	Sets the port B 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_b_pullups(FALSE);`
Example Files:	ex_lcdkb.c with kdb.c
Also See:	input(), input_x(), output_float()

POW( ), PWR( )

Syntax:	f = pow (***x***,***y***) f = pwr (***x***,***y***)
Parameters:	***x*** and ***y*** and of type float
Returns:	A float
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 = (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)**
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. 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, lcdkb.c

Also See: atoi(), puts(), putc(), getc() (for a stream example), RS232 I/O overview

PSP_OUTPUT_FULL( ), PSP_INPUT_FULL( ), PSP_OVERFLOW( )

Syntax:	result = psp_output_full() result = psp_input_full() result = psp_overflow()
Parameters:	None
Returns:	A 0 (FALSE) or 1 (TRUE)
Function:	These functions check the Parallel Slave Port (PSP) for the indicated conditions and return TRUE or FALSE.
Availability:	This function is only available on devices with PSP hardware on chips.
Requires:	Nothing

Examples:	`while (psp_output_full()) ;`
	`psp_data = command;`
	`while(!psp_input_full()) ;`
	`if ( psp_overflow() )`
	`  error = TRUE;`
	`else`
	`  data = psp_data;`

Example Files:	ex_psp.c
Also See:	setup_psp(), PSP overview

PSP_INPUT_FULL( )

See PSP_OUTPUT_FULL

PSP_OVERFLOW( )

See PSP_OUTPUT_FULL

PUTC( ), PUTCHAR( ), FPUTC( )

Syntax:	putc (***cdata***) putchar (***cdata***) value = fputc(***cdata***, ***stream***)
Parameters:	***cdata*** is a 8 bit character. ***Stream*** is a stream identifier (a constant byte)
Returns:	undefined
Function:	This function sends a character over the RS232 XMIT pin. A #USE RS232 must appear before this call to determine the baud rate and pin used. The #USE RS232 remains in effect until another is encountered in the file. If fputc() is used then the specified stream is used where putc() defaults to STDOUT (the last USE RS232).
Availability:	All devices
Requires:	#use rs232

Examples:	`putc('*');`
	`for(i=0; i<10; i++)`
	`  putc(buffer[i]);`
	`putc(13);`

Example Files:	ex_tget.c
Also See:	getc(), printf(), #USE RS232, RS232 I/O overview

PUTCHAR( )

See PUTC

PUTS( ), FPUTS( )

Syntax:	puts (***string***). value = 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 RETURN (13) and LINE-FEED (10) are sent. In general printf() is more useful than puts(). If fputs() is used then the specified stream is used where puts() defaults to STDOUT (the last USE RS232)
Availability:	All devices
Requires:	#use rs232

Examples:	`puts( " ----------- " );`
	`puts( " |   HI    | " );`
	`puts( " ----------- " );`

Example Files:	None
Also See:	printf(), gets(), RS232 I/O overview

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(const void *arg1,const void *arg2);`
	
	`void main()  {`
	`  qsort ( nums, 5, sizeof(int), compare);`
	`}`
	
	`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:	bsearch()

RAND( )

Syntax:	re=rand()
Parameters:	None
Returns:	A pseudo-random integer.
Function:	The rand function returns a sequence of pseudo-random integers in the range of 0 to RAND_MAX.
Availability:	All devices
Requires:	#include <STDLIB.H>
Examples:	`int I;` `I=rand();`
Example Files:	None
Also See:	srand()

READ_ADC( )

Syntax: value = read_adc ([***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	8 bit	10 bit	11 bit	16 bit
ADC=8	00-FF	00-FF	00-FF	00-FF
ADC=10	x	0-3FF	x	x
ADC=11	x	x	0-7FF	x
ADC=16	0-FF00	0-FFC0	0-FFEO	0-FFFF

Note: x is not defined

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

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

Examples:

```
setup_adc(  ADC_CLOCK_INTERNAL  );
setup_adc_ports( ALL_ANALOG );
set_adc_channel(1);
while ( input(PIN_B0) ) {
   delay_ms( 5000 );
   value = read_adc();
   printf("A/D value = %2x\n\r", value);
}

read_adc(ADC_START_ONLY);
sleep();
value=read_adc(ADC_READ_ONLY);
```

Example Files: ex_admm.c, ex_14kad.c

Also See: setup_adc(), set_adc_channel(), setup_adc_ports(), #DEVICE, ADC overview

READ_BANK( )

Syntax:	value = read_bank (***bank***, ***offset***)
Parameters:	***bank*** is the physical RAM bank 1-3 (depending on the device), ***offset*** is the offset into user RAM for that bank (starts at 0),
Returns:	8 bit int
Function:	Read a data byte from the user RAM area of the specified memory bank. This function may be used on some devices where full RAM access by auto variables is not efficient. For example, setting the pointer size to 5 bits on the PIC16C57 chip will generate the most efficient ROM code. However, auto variables can not be about 1Fh. Instead of going to 8 bit pointers, you can save ROM by using this function to write to the hard-to-reach banks. In this case, the bank may be 1-3 and the offset may be 0-15.
Availability:	All devices but only useful on PCB parts with memory over 1Fh and PCM parts with memory over FFh.
Requires:	Nothing

Examples:	`// See write_bank() example to see`
	`// how we got the data`
	`// Moves data from buffer to LCD`
	`i=0;`
	`do {`
	`  c=read_bank(1,i++);`
	`  if(c!=0x13)`
	`  lcd_putc(c);`
	`} while (c!=0x13);`

Example Files:	ex_psp.c
Also See:	write_bank(), and the "Common Questions and Answers" section for more information.

READ_CALIBRATION( )

Syntax:	value = read_calibration (***n***)
Parameters:	***n*** is an offset into calibration memory beginning at 0
Returns:	An 8 bit byte
Function:	The read_calibration function reads location "n" of the 14000-calibration memory.
Availability:	This function is only available on the PIC14000.
Requires:	Nothing
Examples:	fin = read_calibration(16);
Example Files:	ex_14kad.c with ex_14cal.c
Also See:	None

READ_EEPROM( )

Syntax:	value = read_eeprom (***address***)
Parameters:	***address*** is an (8 bit or 16 bit depending on the part) int
Returns:	An 8 bit int
Function:	Reads a byte from the specified data EEPROM address. The address begins at 0 and the range depends on the part.
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:	ex_intee.c
Also See:	write_eeprom(), data eeprom overview

READ_PROGRAM_EEPROM( )

Syntax:	value = read_program_eeprom (***address***)
Parameters:	***address*** is 16 bits on PCM parts and 32 bits on PCH parts
Returns:	16 bits
Function:	Reads data from the program memory.
Availability:	Only devices that allow reads from program memory.
Requires:	Nothing

Examples:	`checksum = 0;`
	`for(i=0;i<8196;i++)`
	`  checksum^=read_program_eeprom(i);`
	`printf("Checksum is %2X\r\n",checksum);`

Example Files:	None
Also See:	write_program_eeprom(), write_eeprom(), read_eeprom() Program eeprom overview

READ_PROGRAM_MEMORY( ), READ_EXTERNAL_MEMORY( )

Syntax:	READ_PROGRAM_MEMORY (***address***, ***dataptr***, ***count***); READ_EXTERNAL_MEMORY (***address***, ***dataptr***, ***count***);
Parameters:	***address*** is 16 bits on PCM parts and 32 bits on PCH parts. The least significant bit should always be 0 in PCM. ***dataptr*** is a pointer to one or more bytes. ***count*** is a 8 bit integer
Returns:	undefined
Function:	Reads count bytes from program memory at address to RAM at dataptr. Both of these functions operate exactly the same.
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( ), External memory overview, Program eeprom overview

READ_EXTERNAL_MEMORY( )

See: Read_Program_Memory

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 byes 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:	STDLIBM.H must be included

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:	None
Also See:	None

RESTART_CAUSE( )

Syntax:	value = restart_cause()
Parameters:	None
Returns:	A value indicating the cause of the last processor reset. The actual values are device dependent. See the device .h file for specific values for a specific device. Some example values are: WDT_FROM_SLEEP, WDT_TIMEOUT, MCLR_FROM_SLEEP and NORMAL_POWER_UP.
Function:	Returns the cause of the last processor reset.
Availability:	All devices
Requires:	Constants are defined in the devices .h file.

Examples:	`switch ( restart_cause() ) {`
	`  case WDT_FROM_SLEEP:`
	`  case WDT_TIMEOUT:`
	`  handle_error();`
	`}`

Example Files:	ex_wdt.c
Also See:	restart_wdt(), reset_cpu()

RESTART_WDT( )

Syntax:	restart_wdt()
Parameters:	None
Returns:	undefined
Function:	Restarts the watchdog timer. If the watchdog timer is enabled, this must be called periodically to prevent the processor from resetting. The watchdog timer is used to cause a hardware reset if the software appears to be stuck. The timer must be enabled, the timeout time set and software must periodically restart the timer. These are done differently on the PCB/PCM and PCH parts as follows: PCB/PCM: Enable/Disable: #fuses; Timeout time: setup_wdt(); restart: restart_wdt() PCH: Enable/Disable: setup_wdt(); Timeout time: #fuses; restart: restart_wdt()
Availability:	All devices
Requires:	#fuses

Examples:	`#fuses WDT  // PCB/PCM example`
	`  // See setup_wdt for a PIC18 example`
	`main() {`
	`  setup_wdt(WDT_2304MS);`
	`  while (TRUE) {`
	`  restart_wdt();`
	`  perform_activity();`
	`  }`
	`}`

Example Files:	ex_wdt.c
Also See:	#fuses, setup_wdt(), WDT or Watch Dog Timer overview

ROTATE_LEFT( )

Syntax:	rotate_left (***address***, ***bytes***)
Parameters:	***address*** is a pointer to memory, ***bytes*** is a count of the number of bytes to work with.
Returns:	undefined
Function:	Rotates a bit through an array or structure. The address may be an array identifier or an address to a byte or structure (such as &data). Bit 0 of the lowest BYTE in RAM is considered the LSB.
Availability:	All devices
Requires:	Nothing

Examples:	`x = 0x86;`
	`rotate_left( &x, 1);`
	`// x is now 0x0d`

Example Files:	None
Also See:	rotate_right(), shift_left(), shift_right()

ROTATE_RIGHT( )

Syntax:	rotate_right (***address***, ***bytes***)
Parameters:	***address*** is a pointer to memory, ***bytes*** is a count of the number of bytes to work with.
Returns:	undefined
Function:	Rotates a bit through an array or structure. The address may be an array identifier or an address to a byte or structure (such as &data). Bit 0 of the lowest BYTE in RAM is considered the LSB.
Availability:	All devices
Requires:	Nothing

Examples:	`struct {`
	`  int cell_1 : 4;`
	`  int cell_2 : 4;`
	`  int cell_3 : 4;`
	`  int cell_4 : 4; } cells;`
	`rotate_right( &cells, 2);`
	`rotate_right( &cells, 2);`
	`rotate_right( &cells, 2);`
	`rotate_right( &cells, 2);`
	`// cell_1->4, 2->1, 3->2 and 4-> 3`

Example Files:	None
Also See:	rotate_left(), shift_left(), shift_right()

SET_ADC_CHANNEL( )

Syntax:	set_adc_channel (***chan***)
Parameters:	***chan*** is the channel number to select. Channel numbers start at 0 and are labeled in the data sheet AN0, AN1
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:	This function is only available on devices with A/D hardware.
Requires:	Nothing

Examples:	`set_adc_channel(2);`
	`delay_us(10);`
	`value = read_adc();`

Example Files:	ex_admm.c
Also See:	read_adc(), setup_adc(), setup_adc_ports(), ADC overview

SET_PWM1_DUTY( ), SET_PWM2_DUTY( ), SET_PWM3_DUTY( ), SET_PWM4_DUTY( ), SET_PWM5_DUTY( )

Syntax:	set_pwm1_duty (***value***) set_pwm2_duty (***value***) set_pwm3_duty (***value***) set_pwm4_duty (***value***) set_pwm5_duty (***value***)
Parameters:	***value*** may be an 8 or 16 bit constant or variable.
Returns:	undefined
Function:	Writes the 10-bit value to the PWM to set the duty. An 8-bit value may be used if the least significant bits are not required. If **value** is an 8 bit item, it is shifted up with two zero bits in the lsb positions to get 10 bits. The 10 bit value is then used to determine the amount of time the PWM signal is high during each cycle as follows: • value*(1/clock)*t2div Where clock is oscillator frequency and t2div is the timer 2 prescaler (set in the call to setup_timer2).
Availability:	This function is only available on devices with CCP/PWM hardware.
Requires:	Nothing

Examples:	`// For a 20 mhz clock, 1.2 khz frequency,`
	`// t2DIV set to 16`
	`// the following sets the duty to 50% (or 416 us).`
	
	`long duty;`
	
	`duty = 512; // .000416/(16*(1/20000000))`
	`set_pwm1_duty(duty);`

Example Files:	ex_pwm.c
Also See:	setup_ccpX(), CCP1 overview

SET_PWM2_DUTY
SET_PWM3_DUTY
SET_PWM4_DUTY
SET_PWM5_DUTY

See SET_PWM1_DUTY

SET_POWER_PWMX_DUTY( )

Syntax:	set_power_pwm***X***_duty(***duty***)
Parameters:	***X*** is 0, 2, 4, or 6 ***Duty*** is an integer between 0 and 16383.
Returns:	undefined
Function:	Stores the value of duty into the appropriate PDCXL/H register. This duty value is the amount of time that the PWM output is in the active state.
Availability:	All devices equipped with PWM.
Requires:	None
Examples:	`set_power_pwm0_duty(4000);`
Example Files:	None
Also See:	setup_power_pwm(), setup_power_pwm_pins(),set_power_pwm_override()

SET_POWER_PWM_OVERRIDE( )

Syntax:	set_power_pwm_override(***pwm***, ***override***, ***value***)
Parameters:	***pwm*** is a constant between 0 and 7 ***Override*** is true or false ***Value*** is 0 or 1
Returns:	undefined
Function:	***pwm*** selects which module will be affected. **Override** determines whether the output is to be determined by the OVDCONS register or the PDC registers. When override is false, the PDC registers determine the output. When override is true, the output is determined by the value stored in OVDCONS. When ***value*** is a 1, the PWM pin will be driven to its active state on the next duty cycle. If value is 0, the pin will be inactive.
Availability:	All devices equipped with PWM.
Requires:	None

Examples:	`set_power_pwm_override(1, true, 1); //PWM1 will be overridden to active state`
	`set_power_pwm_override(1, false, 0); //PMW1 will not be overidden`

Example Files:	None
Also See:	setup_power_pwm(), setup_power_pwm_pins(),set_power_pwmX_duty()

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 & 3 get a 16 bit int. Timer 2 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.
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 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_TIMER0( )
SET_TIMER1( )
SET_TIMER2( )
SET_TIMER3( )
SET_TIMER4( )
SET_TIMER5( )

See SET_RTCC

SET_TRIS_A( ), SET_TRIS_B( ), SET_TRIS_C( ), SET_TRIS_D( ), SET_TRIS_E( ), SET_TRIS_F( ), SET_TRIS_G( ), SET_TRIS_H( ), SET_TRIS_J( ), SET_TRIS_K( )

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 8 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 #BYTE 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`
	`  // B3,B2,B1,B0 are inputs`

Example Files:	lcd.c
Also See:	#USE FAST_IO, #USE FIXED_IO, #USE STANDARD_IO

SET_TRIS_B( )
SET_TRIS_C( )
SET_TRIS_D( )
SET_TRIS_E( )
SET_TRIS_F( )
SET_TRIS_G( )
SET_TRIS_H( )
SET_TRIS_J( )
SET_TRIS_K( )

See SET_TRIS_A

SET_UART_SPEED( )

Syntax:	set_uart_speed (***baud***, [***stream***])
Parameters:	***baud*** is a constant 100-115200 representing the number of bits per second. ***stream*** is an optional stream identifier.
Returns:	undefined
Function:	Changes the baud rate of the built-in hardware RS232 serial port at run-time.
Availability:	This function is only available on devices with a built in UART.
Requires:	#use rs232

Examples:	`// Set baud rate based on setting`
	`// of pins B0 and B1`
	
	`switch( input_b() & 3 ) {`
	`  case 0 : set_uart_speed(2400);  break;`
	`  case 1 : set_uart_speed(4800);  break;`
	`  case 2 : set_uart_speed(9600);  break;`
	`  case 3 : set_uart_speed(19200);  break;`
	`}`

Example Files:	loader.c
Also See:	#USE RS232, putc(), getc(), RS232 I/O overview

SETJMP( )

Syntax:	result = setjmp (***env***)
Parameters:	***env***: The data object that will receive the current environment
Returns:	If the return is from a direct invocation, this function returns 0. If the return is from a call to the longjmp function, the setjmp function returns a nonzero value and it's the same value passed to the longjmp function.
Function:	Stores information on the current calling context in a data object of type jmp_buf and which marks where you want control to pass on a corresponding longjmp call.
Availability:	All devices
Requires:	#include <setjmp.h>
Examples:	`result = setjmp(jmpbuf);`
Example Files:	None
Also See:	longjmp()

SETUP_ADC(mode)

Syntax:	setup_adc (***mode***);
Parameters:	***mode***- Analog to digital mode. The valid options vary depending on the device. See the devices .h file for all options. Some typical options include: • ADC_OFF • ADC_CLOCK_INTERNAL • ADC_CLOCK_DIV_32
Returns:	undefined
Function:	Configures the analog to digital converter.
Availability:	Only the devices with built in analog to digital converter.
Requires:	Constants are defined in the devices .h file.

Examples:	`setup_adc_ports( ALL_ANALOG );`
	`setup_adc(ADC_CLOCK_INTERNAL );`
	`set_adc_channel( 0 );`
	`value = read_adc();`
	`setup_adc( ADC_OFF );`

Example Files:	ex_admm.c
Also See:	setup_adc_ports(), set_adc_channel(), read_adc(), #device. The device .h file., ADC overview

SETUP_ADC_PORTS()

Syntax:	setup_adc_ports (***value***)
Parameters:	***value*** - a constant defined in the devices .h file
Returns:	undefined
Function:	Sets up the ADC pins to be analog, digital or a combination. The allowed combinations vary depending on the chip. The constants used are different for each chip as well. Check the device include file for a complete list. The constants ALL_ANALOG and NO_ANALOGS are valid for all chips. Some other example constants: • ANALOG_RA3_REF- All analog and RA3 is the reference • RA0_RA1_RA3_ANALOG- Just RA0, RA1 and RA3 are analog
Availability:	This function is only available on devices with A/D hardware.
Requires:	Constants are defined in the devices .h file.

Examples:	`// All pins analog (that can be)`
	
	`setup_adc_ports( ALL_ANALOG );`
	
	`// Pins A0, A1 and A3 are analog and all others`
	`// are digital. The +5v is used as a reference.`
	`setup_adc_ports( RA0_RA1_RA3_ANALOG );`
	
	`// Pins A0 and A1 are analog. Pin RA3 is used`
	`// for the reference voltage and all other pins`
	`// are digital.`
	`setup_adc_ports( A0_RA1_ANALOGRA3_REF );`

Example Files:	ex_admm.c
Also See:	setup_adc(), read_adc(), set_adc_channel() , ADC overview

SETUP_CCP1( ), SETUP_CCP2( ), SETUP_CCP3( ), SETUP_CCP4( ), SETUP_CCP5( ), SETUP_CCP6( )

Syntax:

setup_ccp1 (***mode***) or setup_ccp1 (***mode, pwm***)
setup_ccp2 (***mode***) or setup_ccp2 (***mode, pwm***)
setup_ccp3 (***mode***) or setup_ccp3 (***mode, pwm***)
setup_ccp4 (***mode***) or setup_ccp4 (***mode, pwm***)
setup_ccp5 (***mode***) or setup_ccp5 (***mode, pwm***)
setup_ccp6 (***mode***) or setup_ccp6 (***mode, pwm***)

Parameters:

mode is a constant. Valid constants are in the devices .h file and are as follows:

Disable the CCP:

CCP_OFF

Set CCP to capture mode:

CCP_CAPTURE_FE	Capture on falling edge
CCP_CAPTURE_RE	Capture on rising edge
CCP_CAPTURE_DIV_4	Capture after 4 pulses
CCP_CAPTURE_DIV_16	Capture after 16 pulses

Set CCP to compare mode:

CCP_COMPARE_SET_ON_MATCH	Output high on compare
CCP_COMPARE_CLR_ON_MATCH	Output low on compare
CCP_COMPARE_INT	interrupt on compare
CCP_COMPARE_RESET_TIMER	Reset timer on compare

Set CCP to PWM mode:

CCP_PWM	Enable Pulse Width Modulator

pwm parameter is an optional parameter for chips that includes ECCP module. This parameter allows setting the shutdown time. The value may be 0-255.

CCP_PWM_H_H
CCP_PWM_H_L
CCP_PWM_L_H
CCP_PWM_L_L
CCP_PWM_FULL_BRIDGE
CCP_PWM_FULL_BRIDGE_REV
CCP_PWM_HALF_BRIDGE

CCP_SHUTDOWN_ON_COMP1	shutdown on Comparator 1 change
CCP_SHUTDOWN_ON_COMP2	shutdown on Comparator 2 change
CCP_SHUTDOWN_ON_COMP	Either Comp. 1 or 2 change
CCP_SHUTDOWN_ON_INT0	VIL on INT pin
CCP_SHUTDOWN_ON_COMP1_INT0	VIL on INT pin or Comparator 1 change
CCP_SHUTDOWN_ON_COMP2_INT0	VIL on INT pin or Comparator 2 change
CCP_SHUTDOWN_ON_COMP_INT0	VIL on INT pin or Comparator 1 or 2 change
CCP_SHUTDOWN_AC_L	Drive pins A nad C high
CCP_SHUTDOWN_AC_H	Drive pins A nad C low
CCP_SHUTDOWN_AC_F	Drive pins A nad C tri-state
CCP_SHUTDOWN_BD_L	Drive pins B nad D high
CCP_SHUTDOWN_BD_H	Drive pins B nad D low
CCP_SHUTDOWN_BD_F	Drive pins B nad D tri-state
CCP_SHUTDOWN_RESTART	the device restart after a shutdown event
CCP_DELAY	use the dead-band delay

Returns:	undefined
Function:	Initialize the CCP. The CCP counters may be accessed using the long variables CCP_1 and CCP_2. The CCP operates in 3 modes. In capture mode it will copy the timer 1 count value to CCP_x when the input pin event occurs. In compare mode it will trigger an action when timer 1 and CCP_x are equal. In PWM mode it will generate a square wave. The PCW wizard will help to set the correct mode and timer settings for a particular application.
Availability:	This function is only available on devices with CCP hardware.
Requires:	Constants are defined in the devices .h file.
Examples:	`setup_ccp1(CCP_CAPTURE_RE);`
Example	ex_pwm.c, ex_ccpmp.c, ex_ccp1s.c

Files:
Also See: set_pwmX_duty(), CCP1 overview

SETUP_CCP2( )
SETUP_CCP3( )
SETUP_CCP4( )
SETUP_CCP5( )

See: SETUP_CCP1( )

SETUP_COMPARATOR( )

Syntax:	setup_comparator (***mode***)
Parameters:	***mode*** is a constant. Valid constants are in the devices .h file and are as follows: A0_A3_A1_A2 A0_A2_A1_A2 NC_NC_A1_A2 NC_NC_NC_NC A0_VR_A1_VR A3_VR_A2_VR A0_A2_A1_A2_OUT_ON_A3_A4 A3_A2_A1_A2
Returns:	undefined
Function:	Sets the analog comparator module. The above constants have four parts representing the inputs: C1-, C1+, C2-, C2+
Availability:	This function is only available on devices with an analog comparator.
Requires	Constants are defined in the devices .h file.

Examples:	`// Sets up two independent comparators (C1 and C2),`
	`// C1 uses A0 and A3 as inputs (- and +), and C2`
	`// uses A1 and A2 as inputs`
	`setup_comparator(A0_A3_A1_A2);`

Example Files:	ex_comp.c
Also See:	Analog Comparator overview

SETUP_COUNTERS( )

Syntax:	setup_counters (***rtcc_state***, ***ps_state***)
Parameters:	***rtcc_state*** may be one of the constants defined in the devices .h file. For example: RTCC_INTERNAL, RTCC_EXT_L_TO_H or RTCC_EXT_H_TO_L ***ps_state*** may be one of the constants defined in the devices .h file. For example: RTCC_DIV_2, RTCC_DIV_4, RTCC_DIV_8, RTCC_DIV_16, RTCC_DIV_32, RTCC_DIV_64, RTCC_DIV_128, RTCC_DIV_256, WDT_18MS, WDT_36MS, WDT_72MS, WDT_144MS, WDT_288MS, WDT_576MS, WDT_1152MS, WDT_2304MS
Returns:	undefined
Function:	Sets up the RTCC or WDT. The rtcc_state determines what drives the RTCC. The PS state sets a prescaler for either the RTCC or WDT. The prescaler will lengthen the cycle of the indicated counter. If the RTCC prescaler is set the WDT will be set to WDT_18MS. If the WDT prescaler is set the RTCC is set to RTCC_DIV_1. This function is provided for compatibility with older versions. setup_timer_0 and setup_WDT are the recommended replacements when possible. For PCB devices if an external RTCC clock is used and a WDT prescaler is used then this function must be used.
Availability:	All devices
Requires:	Constants are defined in the devices .h file.
Examples:	`setup_counters (RTCC_INTERNAL, WDT_2304MS);`
Example Files:	None
Also See:	setup_wdt(), setup_timer_0(), devices .h file

SETUP_EXTERNAL_MEMORY( )

Syntax:	SETUP_EXTERNAL_MEMORY(***mode***);
Parameters:	***mode*** is one or more constants from the device header file OR'ed together.
Returns:	undefined
Function:	Sets the mode of the external memory bus.
Availability:	Only devices that allow external memory.
Requires:	Device .h file.

Examples:	`setup_external_memory(EXTMEM_WORD_WRITE`
	`|EXTMEM_WAIT_0 );`
	`setup_external_memory(EXTMEM_DISABLE);`

Example Files:	None
Also See:	WRITE_PROGRAM_EEPROM(), WRITE_PROGRAM_MEMORY(), External Memory overview

SETUP_LCD( )

Syntax:	setup_lcd (***mode***, ***prescale***, [***segments***]);
Parameters:	***Mode*** may be one of these constants from the devices .h file: • LCD_DISABLED, LCD_STATIC, LCD_MUX12,LCD_MUX13, LCD_MUX14 The following may be or'ed (via |) with any of the above: • STOP_ON_SLEEP, USE_TIMER_1 See the devices.h file for other device specific options. ***Prescale*** may be 0-15 for the LCD clock**.** ***Segments*** may be any of the following constants or'ed together: • SEGO_4, SEG5_8, SEG9_11, SEG12_15, SEG16_19, SEGO_28, SEG29_31, ALL_LCD_PINS If omitted the compiler will enable all segments used in the program.
Returns:	undefined
Function:	This function is used to initialize the 923/924 LCD controller.
Availability:	Only devices with built in LCD drive hardware.
Requires	Constants are defined in the devices .h file.
Examples:	`setup_lcd(LCD_MUX14|STOP_ON_SLEEP,2);`
Example Files:	ex_92lcd.c
Also See:	lcd_symbol(), lcd_load(), Internal LCD overview

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
Example Files:	None
Also See:	None

SETUP_OSCILLATOR( )

Syntax:	setup_oscillator(***mode***, ***finetune***)
Parameters:	***mode*** is dependent on the chip. For example, some chips allow speed setting such as OSC_8MHZ or OSC_32KHZ. Other chips permit changing the source like OSC_TIMER1. The ***finetune*** (only allowed on certain parts) is a signed int with a range of -31 to +31.
Returns:	Some chips return a state such as OSC_STATE_STABLE to indicate the oscillator is stable.
Function:	This function controls and returns the state of the internal RC oscillator on some parts. See the devices .h file for valid options for a particular device. Note that if INTRC or INTRC_IO is specified in #fuses and a #USE DELAY is used for a valid speed option, then the compiler will do this setup automatically at the start of main(). WARNING: If the speed is changed at run time the compiler may not generate the correct delays for some built in functions. The last #USE DELAY encountered in the file is always assumed to be the correct speed. You can have multiple #USE DELAY lines to control the compilers knowledge about the speed.
Availability:	Only parts with a OSCCON register.
Requires:	Constants are defined in the .h file.
Examples:	`setup_oscillator( OSC_2MHZ );`
Example Files:	None
Also See:	#fuses, Internal oscillator overview

SETUP_OPAMP1( )
SETUP_OPAMP2( )

Syntax:	setup_opamp1(***enabled***) setup_opamp2(***enabled***)
Parameters:	***enabled*** can be either TRUE or FALSE.
Returns:	undefined
Function:	Enables or Disables the internal operational amplifier peripheral of certain PICmicros.
Availability:	Only parts with a built-in operational amplifier (for example, PIC16F785).
Requires:	Only parts with a built-in operational amplifier (for example, PIC16F785).
Examples:	`setup_opamp1(TRUE);` `setup_opamp2(boolean_flag);`
Example Files:	None
Also See:	None

SETUP_OPAMP2( )

See: SETUP_OPAMP1( )

SETUP_POWER_PWM( )

Syntax: setup_power_pwm(***modes***, ***postscale***, ***time_base***, ***period***, ***compare***, ***compare_postscale***, ***dead_time***)

Parameters: ***modes*** values may be up to one from each group of the following:
PWM_CLOCK_DIV_4, PWM_CLOCK_DIV_16,
PWM_CLOCK_DIV_64, PWM_CLOCK_DIV_128

PWM_OFF, PWM_FREE_RUN, PWM_SINGLE_SHOT,
PWM_UP_DOWN, PWM_UP_DOWN_INT

PWM_OVERRIDE_SYNC

PWM_UP_TRIGGER,

PWM_DOWN_TRIGGER
PWM_UPDATE_DISABLE, PWM_UPDATE_ENABLE

PWM_DEAD_CLOCK_DIV_2,
PWM_DEAD_CLOCK_DIV_4,
PWM_DEAD_CLOCK_DIV_8,
PWM_DEAD_CLOCK_DIV_16

postscale is an integer between 1 and 16. This value sets the PWM time base output postscale.

time_base is an integer between 0 and 65535. This is the initial value of the PWM base

timer.period is an integer between 0 and 4095. The PWM time base is incremented until it reaches this number.

compare is an integer between 0 and 255. This is the value that the PWM time base is compared to, to determine if a special event should be triggered.

compare_postscale is an integer between 1 and 16. This postscaler affects compare, the special events trigger.

dead_time is an integer between 0 and 63. This value specifies the length of an off period that should be inserted between the going off of a pin and the going on of it is a complementary pin.

Returns: undefined

Function:	Initializes and configures the Pulse Width Modulation (PWM) device.
Availability:	All devices equipped with PWM.
Requires:	None
Examples:	`setup_power_pwm(PWM_CLOCK_DIV_4 | PWM_FREE_RUN | PWM_DEAD_CLOCK_DIV_4,1,10000,1000,0,1,0);`
Example Files:	None
Also See:	set_power_pwm_override(), setup_power_pwm_pins(), set_power_pwmX_duty()

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 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 Files:	None
Also See:	setup_power_pwm(), set_power_pwm_override(),set_power_pwmX_duty()

SETUP_PSP( )

Syntax:	setup_psp (***mode***)
Parameters:	***mode*** may be: PSP_ENABLED PSP_DISABLED
Returns:	undefined
Function:	Initializes the Parallel Slave Port (PSP). The SET_TRIS_E(value) function may be used to set the data direction. The data may be read and written to using the variable PSP_DATA.
Availability:	This function is only available on devices with PSP hardware.
Requires:	Constants are defined in the devices .h file.
Examples:	`setup_psp(PSP_ENABLED);`
Example Files:	ex_psp.c
Also See:	set_tris_e(), PSP overview

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 • Constants from each group may be or'ed together with |.
Returns:	undefined
Function:	Initializes the Serial Port Interface (SPI). This is used for 2 or 3 wire serial devices that follow a common clock/data protocol.
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_clk_div_16 );`
Example Files:	ex_spi.c
Also See:	spi_write(), spi_read(), spi_data_is_in(), SPI 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:	ex_stwt.c
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:	ex_patg.c
Also See:	get_timer1(), set_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:	ex_pwm.c
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, 3_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 PIC®18 devices.
Requires:	Constants are defined in the devices .h file.
Examples:	`setup_timer_3 (T3_INTERNAL | T3_DIV_BY_2);`
Example Files:	None
Also See:	get_timer3(), set_timer3()

SETUP_TIMER_4( )

Syntax:	setup_timer_4 (mode, period, postscale)
Parameters:	***mode*** may be one of: • T4_DISABLED, T4_DIV_BY_1, T4_DIV_BY_4, T4_DIV_BY_16 ***period*** is a int 0-255 that determines when the clock value is reset, ***postscale*** is a number 1-16 that determines how many timer overflows before an interrupt: (1 means once, 2 means twice, and so on).
Returns:	undefined
Function:	Initializes timer 4. The mode specifies the clock divisor (from the oscillator clock). The timer value may be read and written to using GET_TIMER4() and SET_TIMER4(). Timer 4 is a 8 bit counter/timer.
Availability:	This function is only available on devices with timer 4 hardware.
Requires:	Constants are defined in the devices .h file

Examples:	`setup_timer_4 ( T4_DIV_BY_4, 0xc0, 2);`
	`// At 20mhz, the timer will increment every 800ns,`
	`// will overflow every 153.6us,`
	`// and will interrupt every 307.2us.`

Example Files:	ex_pwm.c
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 PIC®18 devices.
Requires:	Constants are defined in the devices .h file.
Examples:	`setup_timer_5 (T5_INTERNAL | T5_DIV_BY_2);`
Example Files:	None
Also See:	get_timer5(), set_timer5(), Timer5 overview

SETUP_UART( )

Syntax:	setup_uart(***baud***, ***stream***) setup_uart(***baud***)
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
Returns:	undefined
Function:	Very similar to SET_UART_SPEED. If 1 is passed as a parameter, the UART is turned on, and if 0 is passed, UART is turned off. If a BAUD rate is passed to it, the UART is also turned on, if not already on.
Availability:	This function is only available on devices with a built in UART.
Requires:	#use rs232
Examples:	`setup_uart(9600);` `setup_uart(9600, rsOut);`
Example Files:	None
Also See:	#USE RS232, putc(), getc(), RS232 I/O overview

SETUP_VREF( )

Syntax:	setup_vref (***mode*** | ***value***)
Parameters:	***mode*** may be one of the following constants: • FALSE (off) • VREF_LOW for VDD*VALUE/24 • VREF_HIGH for VDD*VALUE/32 + VDD/4 • any may be or'ed with VREF_A2. ***value*** is an int 0-15.
Returns:	undefined
Function:	Establishes the voltage of the internal reference that may be used for analog compares and/or for output on pin A2.
Availability:	This function is only available on devices with VREF hardware.
Requires:	Constants are defined in the devices .h file.
Examples:	`setup_vref (VREF_HIGH | 6);` `// At VDD=5, the voltage is 2.19V`
Example Files:	ex_comp.c
Also See:	Voltage Reference overview

SETUP_WDT( )

Syntax:	setup_wdt (***mode***)
Parameters:	For PCB/PCM parts: WDT_18MS, WDT_36MS, WDT_72MS, WDT_144MS,WDT_288MS, WDT_576MS, WDT_1152MS, WDT_2304MS For PIC®18 parts: WDT_ON, WDT_OFF For PIC®16 parts with software controlled WDT enabled: WDT_ON, WDT_OFF, WDT_TIMES_32, WDT_TIMES_64, WDT_TIMES_128, WDT_TIMES_256, WDT_TIMES_512, WDT_TIMES_1024, WDT_TIMES_2048, WDT_TIMES_4096, WDT_TIMES_8192, WDT_TIMES_16384, WDT_TIMES_32768, WDT_TIMES_65536.
Returns:	undefined
Function:	Sets up the watchdog timer. The watchdog timer is used to cause a hardware reset if the software appears to be stuck. The timer must be enabled, the timeout time set and software must periodically restart the timer. These are done differently on the PCB/PCM and PCH parts as follows: Enable/Disable: PCB/PCM #fuses; PCH #fuses Timeout time: PCB/PCM setup_wdt(); PCH #fuses restart: PCB/PCM restart_wdt(); PCH restart_wdt() Note: For PCH parts and PCM parts wth software controlled WDT, setup_wdt( ) would enable/disable watchdog timer only if NOWDT fuse is set. If WDT fuse is set, watchdog timer is alwasy enabled.
Availability:	All devices
Requires:	#fuses, Constants are defined in the devices .h file.

Examples:	`#fuses WDT1, WDT // PIC18 example, See`
	`// restart_wdt for a PIC18 example`
	`main() { // WDT1 means 18ms*1 for old PIC18s and`
	`// 4ms*1 for new PIC18s`
	`setup_wdt(WDT_ON);`
	`while (TRUE) {`
	`restart_wdt();`
	`perform_activity();`
	`}`
	`}`

Example Files:	ex_wdt.c
Also See:	#fuses, restart_wdt(), WDT or Watch Dog Timer 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
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:	None
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 a float representing an angle in Radians -2pi to 2pi. ***val*** is a float with the range -1.0 to 1.0. ***Value*** is a float.

Returns:

rad is a float representing an angle in Radians -pi/2 to pi/2

val is a float with the range -1.0 to 1.0.

rad1 is a float representing an angle in Radians 0 to pi

rad2 is a float representing an angle in Radians -pi to pi
Result is a float

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:	math.h must be included.

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()

SINH( )

See: SIN()

SLEEP( )

Syntax:	sleep()
Parameters:	None
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:	`SLEEP();`
Example Files:	Ex_wakup.c
Also See:	reset_cpu()

SLEEP_ULPWU( )

Syntax:	sleep_ulpwu(***time***)
Parameters:	***time*** specifies how long, in us, to charge the capacitor on the ultra-low power wakeup pin (by outputting a high on PIN_A0).
Returns:	Undefined
Function:	Charges the ultra-low power wake-up capacitor on PIN_A0 for ***time*** microseconds, and then puts the PIC to sleep. The PIC will then wake-up on an 'Interrupt-on-Change' after the charge on the cap is lost.
Availability:	Ultra Low Power Wake-Up support on the PIC (example, PIC12F683)

Requires:	#use delay

Examples:	`while(TRUE)`
	`{`
	`if (input(PIN_A1))`
	`//do something`
	`else`
	`sleep_ulpwu(10); //cap will be charged for 10us, then goto sleep`
	`}`

Example Files:	None
Also See:	#use delay

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_READ( ), SPI_READ2( )

Syntax:	value = spi_read (***data***) value = spi_read2 (***data***)
Parameters:	***data*** is optional 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 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. 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:	`in_data = spi_read(out_data);`
Example Files:	ex_spi.c
Also See:	spi_data_is_in(), spi_write(), SPI overview

SPI_WRITE( ), SPI_WRITE2( )

Syntax:	SPI_WRITE (***value***) SPI_WRITE2 (***value***)
Parameters:	***value*** is an 8 bit int
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_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

SPRINTF( )

Syntax:	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.
Returns:	Nothing
Function:	This function operates like printf except that the output is placed into the specified string. The output string will be terminated with a null. No checking is done to ensure the string is large enough for the data. See printf() for details on formatting.
Availability:	All devices.
Requires:	Nothing

Examples:	`char mystring[20];`
	`long mylong;`
	
	`mylong=1234;`
	`sprintf(mystring,"<%lu>",mylong);`
	`// mystring now has:`
	`//  < 1 2 3 4 > \0`

Example Files:	None
Also See:	printf()

SQRT( )

Syntax:	result = sqrt (***value***)
Parameters:	***value*** is a float
Returns:	A float
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( sqr(x1-x2) + sqr(y1-y2) );`
Example Files:	None
Also See:	None

SRAND( )

Syntax:	srand(***n***)
Parameters:	***n*** is the seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand.
Returns:	No value.
Function:	The srand function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand. If srand is then called with same seed value, the sequence of random numbers shall be repeated. If rand is called before any call to srand have been made, the same sequence shall be generated as when srand is first called with a seed value of 1.
Availability:	All devices.
Requires:	#include <STDLIB.H>
Examples:	`srand(10);` `I=rand();`
Example Files:	None
Also See:	rand()

STANDARD STRING FUNCTIONS( )

MEMCHR( )
MEMCMP( )
STRCAT( )
STRCHR( )
STRCMP( )
STRCOLL( )
STRCSPN( )
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.
	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]

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:	ex_str.c
Also See:	strcpy(), strtok()

STRCAT( )
STRCHR( )

STRCMP( )
STRCOLL( )

See: STANDARD STRING FUNCTIONS( )

STRCPY( ), STRCOPY( )

Syntax:	strcpy (***dest***, ***src***) strcopy (***dest***, ***src***)
Parameters:	***dest*** is a pointer to a RAM array of characters. ***src*** may be either a pointer to a RAM array of characters or it may be a constant string.
Returns:	undefined
Function:	Copies a constant or RAM string to a RAM string. Strings are terminated with a 0.
Availability:	All devices.
Requires:	Nothing

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

Example Files:	ex_str.c
Also See:	strxxxx()

STRCSPN( )
STRLEN( )
STRLWR( )
STRNCAT( )
STRNCMP( )
STRNCPY( )
STRPBRK( )
STRRCHR( )
STRSPN( )

See: STANDARD STRING FUNCTIONS( )

STRTOD( )

Syntax:	result=strtod(***nptr***,& ***endptr***)
Parameters:	***nptr*** and ***endptr*** are strings
Returns:	result is a float. 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:	STDLIB.H must be included

Examples:	`float result;`
	`char str[12]="123.45hello";`
	`char *ptr;`
	`result=strtod(str,&ptr);`
	`//result is 123.45 and ptr is "hello"`

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

STRTOK( )

Syntax:	ptr = strtok(***s1***, ***s2***)
Parameters:	***s1*** and ***s2*** are pointers to an array of characters (or the name of an array). Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi"). s1 may be 0 to indicate a continue operation.
Returns:	ptr points to a character in s1 or is 0
Function:	Finds next token in s1 delimited by a character from separator string s2 (which can be different from call to call), and returns pointer to it. First call starts at beginning of s1 searching for the first character NOT contained in s2 and returns null if there is none are found. If none are found, it is the start of first token (return value). Function then searches from there for a character contained in s2. If none are found, current token extends to the end of s1, and subsequent searches for a token will return null. If one is found, it is overwritten by '\0', which terminates current token. Function saves pointer to following character from which next search will start. Each subsequent call, with 0 as first argument, starts searching from the saved pointer.
Availability:	All devices.
Requires:	#include <string.h>

Examples:	`char string[30], term[3], *ptr;`
	
	`strcpy(string,"one,two,three;");`
	`strcpy(term,",;");`
	
	`ptr = strtok(string, term);`
	`while(ptr!=0) {`
	`  puts(ptr);`
	`  ptr = strtok(0, term);`
	`  }`
	`  // Prints:`
	`  one`
	`  two`
	`  three`

Example Files:	ex_str.c
Also See:	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:	STDLIB.H must be included

Examples:	`signed long result;`
	`char str[9]="123hello";`
	`char *ptr;`
	`result=strtol(str,&ptr,10);`
	`//result is 123 and ptr is "hello"`

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

STRTOUL( )

Syntax:	result=strtoul(***nptr***,& ***endptr***, ***base***)
Parameters:	***nptr*** and ***endptr*** are strings and ***base*** is an integer
Returns:	result is an unsigned long int. returns the converted value in result , if any. If no conversion could be performed, zero is returned.
Function:	The strtoul function converts the initial portion of the string pointed to by nptr to a long int representation in some radix determined by the value of base. The part of the string after conversion is stored in the object pointed to endptr, provided that endptr is not a null pointer. If nptr is empty or does not have the expected form, no conversion is performed and the value of nptr is stored in the object pointed to by endptr, provided endptr is not a null pointer.
Availability:	All devices.
Requires:	STDLIB.H must be included

Examples:	`long result;`
	`char str[9]="123hello";`
	`char *ptr;`
	`result=strtoul(str,&ptr,10);`
	`//result is 123 and ptr is "hello"`

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

STRXFRM( )

See: STANDARD STRING FUNCTIONS( )

SWAP( )

Syntax:	swap (***lvalue***)
Parameters:	***lvalue*** is a byte variable
Returns:	undefined - WARNING: this function does not return the result
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`

Example Files:	None
Also See:	rotate_right(), rotate_left()

TAN( )
TANH( )

See: SIN()

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 Files:	ex_str.c
Also See:	None

WRITE_BANK( )

Syntax:	write_bank (***bank***, ***offset***, ***value***)
Parameters:	***bank*** is the physical RAM bank 1-3 (depending on the device), ***offset*** is the offset into user RAM for that bank (starts at 0), ***value*** is the 8 bit data to write
Returns:	undefined
Function:	Write a data byte to the user RAM area of the specified memory bank. This function may be used on some devices where full RAM access by auto variables is not efficient. For example on the PIC16C57 chip setting the pointer size to 5 bits will generate the most efficient ROM code however auto variables can not be above 1Fh. Instead of going to 8 bit pointers you can save ROM by using this function to write to the hard to reach banks. In this case the bank may be 1-3 and the offset may be 0-15.
Availability:	All devices but only useful on PCB parts with memory over 1Fh and PCM parts with memory over FFh.
Requires:	Nothing

Examples:	`i=0;  // Uses bank 1 as a RS232 buffer`
	`do {`
	`  c=getc();`
	`  write_bank(1,i++,c);`
	`} while (c!=0x13);`

Example Files:	ex_psp.c
Also See:	See the "Common Questions and Answers" section for more information.

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***)
Parameters:	***address*** is a (8 bit or 16 bit depending on the part) int, the range is device dependent, ***value*** is an 8 bit int
Returns:	undefined
Function:	Write a byte to the specified data EEPROM address. This function may take several milliseconds to execute. This works only on devices with EEPROM built into the core of the device. For devices with external EEPROM or with a separate EEPROM in the same package (line the 12CE671) see EX_EXTEE.c with CE51X.c, CE61X.c or CE67X.c.
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:	ex_intee.c, ex_extee.c, ex_ce51x.c, ex_ce61.c, ex_ce67.c
Also See:	read_eeprom(), write_program_eeprom(), read_program_eeprom(), data eeprom overview

WRITE_EXTERNAL_MEMORY( )

Syntax:	write_external_memory(***address***, ***dataptr***, ***count***)
Parameters:	address is 16 bits on PCM parts and 32 bits on PCH parts dataptr is a pointer to one or more bytes count is a 8 bit integer
Returns:	undefined
Function:	Writes count bytes to program memory from dataptr to address. Unlike WRITE_PROGRAM_EEPROM and READ_PROGRAM_EEPROM this function does not use any special EEPROM/FLASH write algorithm. The data is simply copied from register address space to program memory address space. This is useful for external RAM or to implement an algorithm for external flash.
Availability:	Only PCH devices.
Requires:	Nothing

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

Example Files:	None
Also See:	None

WRITE_PROGRAM_EEPROM( )

Syntax:	write_program_eeprom (***address***, ***data***)
Parameters:	***address*** is 16 bits on PCM parts and 32 bits on PCH parts, ***data*** is 16 bits. The least significant bit should always be 0 in PCH.
Returns:	undefined
Function:	Writes to the specified program EEPROM area. See our WRITE_PROGRAM_MEMORY for more information on this function.
Availability:	Only devices that allow writes to program memory.
Requires:	Nothing
Examples:	`write_program_eeprom(0,0x2800);  //disables program`
Example Files:	ex_loadc.c, loader.c
Also See:	read_program_eeprom(), read_eeprom(), write_eeprom(), write_program_memory(), erase_program_eeprom(), Program eeprom overview

WRITE_PROGRAM_MEMORY( )

Syntax:	write_program_memory(***address***, ***dataptr***, ***count***);
Parameters:	***address*** is 16 bits on PCM parts and 32 bits on PCH parts. ***dataptr*** is a pointer to one or more bytes count is a 8 bit integer
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. 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.
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);`
	`}`

Example Files:	loader.c
Also See:	write_program_eeprom, erase_program_eeprom, Program eeprom overview
Additional Notes:	Clarification about the functions to write to program memory: For chips where getenv("FLASH_ERASE_SIZE") > getenv("FLASH_WRITE_SIZE") WRITE_PROGRAM_EEPROM Writes 2 bytes, does not erase (use ERASE_PROGRAM_EEPROM) WRITE_PROGRAM_MEMORY Writes any number of bytes, will erase a block whenever the first (lowest) byte in a block is written to. If the first address is not the start of a block that block is not erased. ERASE_PROGRAM_EEPROM Will erase a block. The lowest address bits are not used. For chips where getenv("FLASH_ERASE_SIZE") = ("FLASH_WRITE_SIZE") WRITE_PROGRAM_EEPROM Writes 2 bytes, no erase is needed. WRITE_PROGRAM_MEMORY Writes any number of bytes, bytes outside the range of the write block are not changed. No erase is needed. ERASE_PROGRAM_EEPROM Not available

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 floating point significant

DBL_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.
DBL_MIN_EXP:	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number.
DBL_MIN_10_EXP:	Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.
DBL_MAX_EXP:	Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number.
DBL_MAX_10_EXP:	Maximum negative integer such that 10 raised to that power is in the range of representable finite floating-point 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 floating point number.
LDBL_MANT_DIG:	Number of base digits in the floating point significant
LDBL_DIG:	Number of decimal digits, q, such that any floating point number with q decimal digits can be rounded into a floating point number with p radix b digits and back again without change to the q decimal digits.
LDBL_MIN_EXP:	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number.
LDBL_MIN_10_EXP:	Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.
LDBL_MAX_EXP:	Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number.
LDBL_MAX_10_EXP:	Maximum negative integer such that 10 raised to that power is in the range of representable finite floating-point numbers.
LDBL_MAX:	Maximum representable finite floating point number.
LDBL_EPSILON:	The difference between 1 and the least value greater than 1 that is representable in the given floating point type.
LDBL_MIN:	Minimum normalized positive floating point number.

limits.h

limits.h	
CHAR_BIT:	Number of bits for the smallest object that is not a bit_field.
SCHAR_MIN:	Minimum value for an object of type signed char
SCHAR_MAX:	Maximum value for an object of type signed char
UCHAR_MAX:	Maximum value for an object of type unsigned char
CHAR_MIN:	Minimum value for an object of type char(unsigned)
CHAR_MAX:	Maximum value for an object of type char(unsigned)
MB_LEN_MAX:	Maximum number of bytes in a multibyte character.
SHRT_MIN:	Minimum value for an object of type short int
SHRT_MAX:	Maximum value for an object of type short int
USHRT_MAX:	Maximum value for an object of type unsigned short int
INT_MIN:	Minimum value for an object of type signed int
INT_MAX:	Maximum value for an object of type signed int
UINT_MAX:	Maximum value for an object of type unsigned int
LONG_MIN:	Minimum value for an object of type signed long int
LONG_MAX:	Maximum value for an object of type signed long int
ULONG_MAX:	Maximum value for an object of type unsigned long int

locale.h

locale.h	
locale.h	(Localization not supported)
lconv	localization structure
SETLOCALE()	returns null
LOCALCONV()	returns clocale

setjmp.h

setjmp.h	
jmp_buf:	An array used by the following functions
setjmp:	Marks a return point for the next longjmp
longjmp:	Jumps to the last marked point

stddef.h

stddef.h	
ptrdiff_t:	The basic type of a pointer
size_t:	The type of the sizeof operator (int)
wchar_t	The type of the largest character set supported (char) (8 bits)
NULL	A null pointer (0)

stdio.h

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

stdlib.h

stdlib.h	
div_t	structure type that contains two signed integers(quot and rem).
ldiv_t	structure type that contains two signed longs(quot and rem
EXIT_FAILURE	returns 1
EXIT_SUCCESS	returns 0
RAND_MAX-	
MBCUR_MAX-	1
SYSTEM()	Returns 0(not supported)
Multibyte character and string 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.

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.

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.

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.

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.

http://www.ccsinfo.com/pic.html

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.

Duplicate USE
Some USE libraries may only be invoked once since they apply to the entire program such as #USE DELAY. These may not be changed throughout the 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 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.

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 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:
```
#xxxxx
#PRAGMA xxxxx
```

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 type conversion

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 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 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.

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 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 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 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
```

Function not void and does not return a value.

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) )
```

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.
```

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 in the PIC®, 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.

Note that in the implementation of the above table, a function call is made when a table is accessed with a subscript that cannot be evaluated at compile time.

How can I pass a variable to functions like OUTPUT_HIGH()?

The pin argument for built in functions like OUTPUT_HIGH need to be known at compile time so the compiler knows the port and bit to generate the correct code.

If your application needs to use a few different pins not known at compile time consider:

```
switch(pin_to_use) {
    case PIN_B3  : output_high(PIN_B3); break;
    case PIN_B4  : output_high(PIN_B4); break;
    case PIN_B5  : output_high(PIN_B5); break;
    case PIN_A1  : output_high(PIN_A1); break;
    }
```

If you need to use any pin on a port use:

```
#byte portb = 6
#byte portb_tris = 0x86  // **

portb_tris &= ~(1<<bit_to_use);  // **

portb |= (1<<bit_to_use);  // bit_to_use is 0-7
```

If you need to use any pin on any port use:

```
*(pin_to_use/8|0x80) &= ~(1<<(pin_to_use&7)); // **

*(pin_to_use/8) |= (1<<(pin_to_use&7));
```

In all cases pin_to_use is the normal PIN_A0... defines.

** These lines are only required if you need to change the direction register (TRIS).

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).

```
#USE  RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1)
void  put_to_a( char c ) {
   put(c);
}
char  get_from_a( ) {
    return(getc()); }
#USE RS232(BAUD=9600, XMIT=PIN_B2,RCV=PIN_B3)
void put_to_b( char b ) {
   putc(c);
}
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:

```
#USE RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1, STREAM=COM_A)
#USE 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 can the RB interrupt be used to detect a button press?

The RB interrupt will happen when there is any change (input or output) on pins B4-B7. There is only one interrupt and the PIC® does not tell you which pin changed. The programmer must determine the change based on the previously known value of the port. Furthermore, a single button press may cause several interrupts due to bounce in the switch. A debounce algorithm will need to be used. The following is a simple example:

```
#int_rb
rb_isr() {
    byte changes;
    changes = last_b ^ port_b;
    last_b = port_b;
    if (bit_test(changes,4 )&& !bit_test(last_b,4)){
         //b4 went low
    }
    if (bit_test(changes,5)&& !bit_test (last_b,5)){
        //b5 went low
    }
    .
    .
    .
    delay_ms (100);  //debounce
}
```

The delay=ms (100) is a quick and dirty debounce. In general, you will not want to sit in an ISR for 100 MS to allow the switch to debounce. A more elegant solution is to set a timer on the first interrupt and wait until the timer overflows. Do not process further changes on the pin.

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 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:

```
#BYTE timer0 = 0x01
timer0= 128; //set timer0 to 128
while (timer0 ! = 200); // wait for timer0 to reach 200
```

Bits in registers may also be mapped as follows:

```
#BIT T0IF = 0x0B.2
.
.
.
while (!T0IF); //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_A(0); rather than *0x85=0;

How do I get getc() to timeout after a specified time?

GETC will always wait for the character to become available. The trick is to not call getc() until a character is ready. This can be determined with kbhit().

The following is an example of how to time out of waiting for an RS232 character.

Note that without a hardware UART the delay_us should be less than a tenth of a bit time (10 us at 9600 baud). With hardware you can make it up to 10 times the bit time. (1000 us at 9600 baud). Use two counters if you need a timeout value larger than 65535.

```
short timeout_error;

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);
   }
}
```

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 put a NOP at location 0 for the ICD?

The CCS compilers are fully compatible with Microchips ICD debugger using MPLAB. In order to prepare a program for ICD debugging (NOP at location 0 and so on) you need to add a #DEVICE ICD=TRUE after your normal #DEVICE.

For example:

```
#INCLUDE <16F877.h>
#DEVICE  ICD=TRUE
```

How do I write variables to EEPROM that are not a byte?

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 type.

- n is an offset into the eeprom.
- For floats you must increment it by 4.
- For example, if the first float is at 0 the second one should be at 4 and the third at 8.

```
WRITE_FLOAT_EXT_EEPROM(long int n, float data) {
   int i;
   for (i = 0; i < 4; i++)
     write_ext_eeprom(i + n, *(&data + i) ) ;
}

float READ_FLOAT_EXT_EEPROM(long int n) {
   int i;
   float data;
   for (i = 0; i < 4; i++)
     *(&data + i) = read_ext_eeprom(i + n);
   return(data);
}
```

How does one map a variable to an I/O port?

Two methods are as follows:

```
#byte   PORTB  =  6   //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;
    int enable : 1;
    int reset  : 1; };
struct    port_b_layout  port_b;
#byte port_b  = 6
struct port_b_layout  const  INIT_1  = {0, 1,1,1,1};
struct port_b_layout  const  INIT_2  = {3, 1,1,1,0};
struct port_b_layout  const  INIT_3  = {0, 0,0,0,0};
struct port_b_layout  const  FOR_SEND = {0,0,0,0,0};
                                  // All outputs
struct    port_b_layout  const  FOR_READ = {15,0,0,0,0};
                                  // 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
    port_b.enable=0;             // independently.
    x = port_b.data;
    port_b.enable=0
}
```

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.

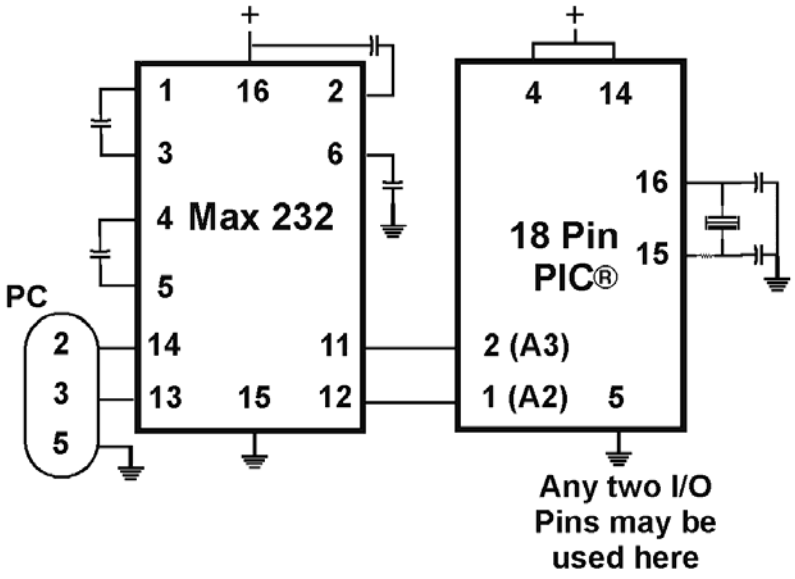
When expressions are converted to relational expressions or SHORT INTs, the result will be FALSE (or 0) when the expression is 0, otherwise the result is TRUE (or 1).

For example:

```
bytevar = 54;
bitvar = bytevar;     //bitvar will be 1 (bytevar ! = O)
if(bytevar)           //will be TRUE
bytevar = 0;
bitvar = bytevar;     //bitvar will be 0
```

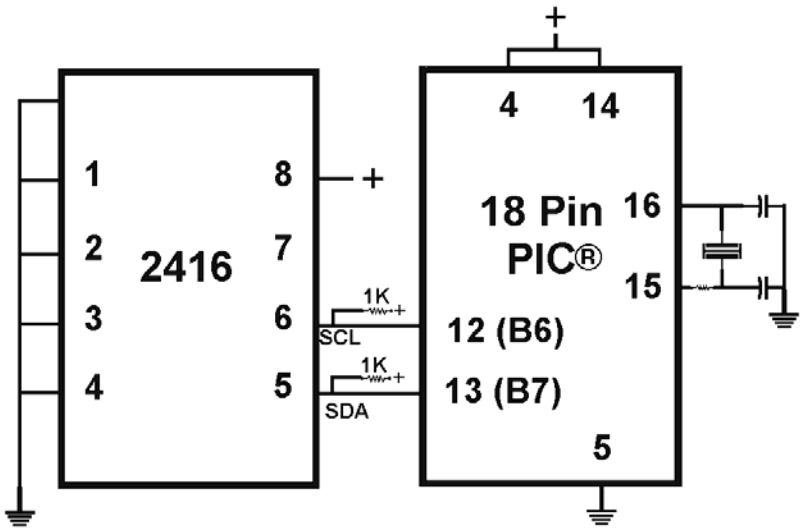
How does the PIC® connect to a PC?

A level converter should be used to convert the TTL (0-5V_ 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 20 mhz, 14 bit PIC®. Note times will vary depending on memory banks used.

20 mhz PIC16

	int8 [us]	int16 [us]	int32 [us]	float [us]
+	0.6	1.4	3	111.3
-	0.6	1.4	3	113.9
*	11.1	47.2	132	178.3
/	23.2	70.8	239.2	330.9
exp()	*	*	*	1697.3
ln()	*	*	*	2017.7
sin()	*	*	*	2184.5

40 mhz PIC18

	int8 [us]	int16 [us]	int32 [us]	float [us]
+	0.3	0.4	0.6	51.3
-	0.3	0.4	0.6	52.3
*	0.4	3.2	22.2	35.8
/	11.3	32	106.6	144.9
exp()	*	*	*	510.4
ln()	*	*	*	644.8
sin()	*	*	*	698.7

Instead of 800, the compiler calls 0. Why?

The PIC® ROM address field in opcodes is 8-10 Bits depending on the chip and specific opcode. The rest of the address bits come from other sources. For example, on the 174 chip to call address 800 from code in the first page will show:

```
BSF    0A,3
CALL   0
```

The call 0 is actually 800H since Bit 11 of the address (Bit 3 of PCLATH, Reg 0A) has been set.

Instead of A0, the compiler is using register 20. Why?

The PIC® RAM address field in opcodes is 5-7 bits long, depending on the chip. The rest of the address field comes from the status register. For example, on the 74 chip to load A0 into W note the following:

```
BSF 3,5
MOVFW     20
```

Note that the BSF may not be immediately before the access since the compiler optimizes out the redundant bank switches.

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

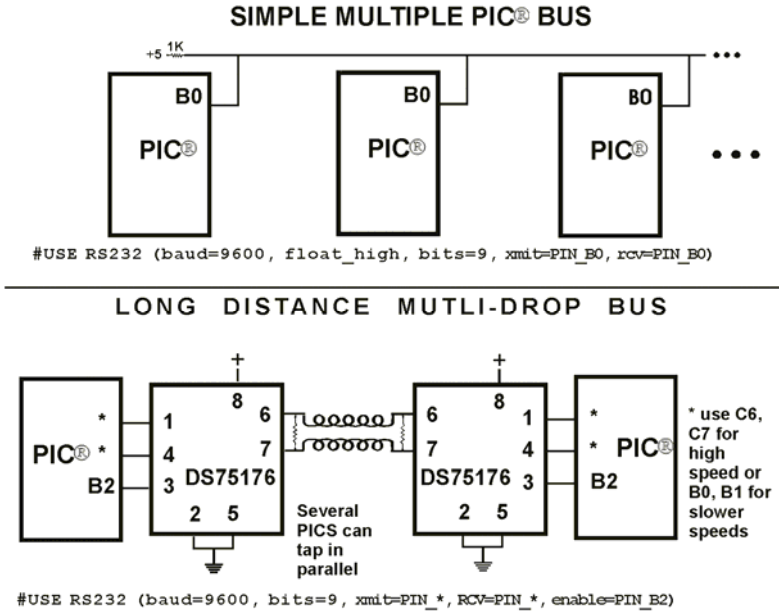
function called by the parent. Sequential calls to functions each with their own local variables is very efficient use of RAM as opposed to a large function with local variables declared for the entire process at once.

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 using 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 two 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.

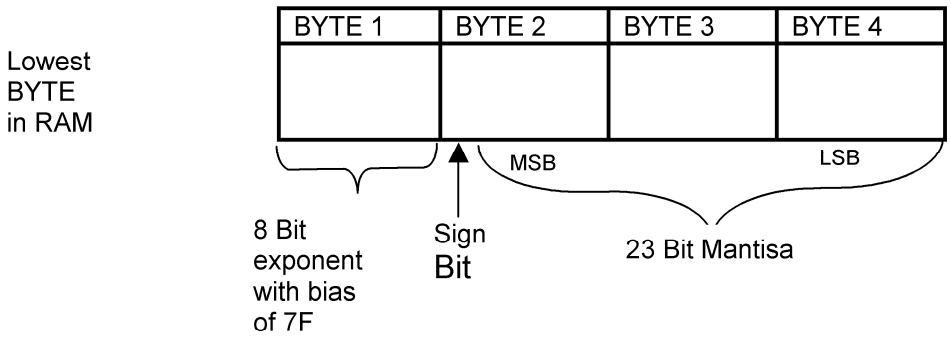
What is an easy way for two or more PICs® to communicate?

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.

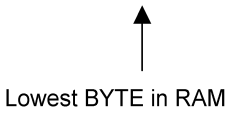


What is the format of floating point numbers?

CCS uses the same format Microchip uses in the 14000 calibration constants. PCW users have a utility Numeric Converter that will provide easy conversion to/from decimal, hex and float in a small window in the Windows IDE. See EX_FLOAT.C for a good example of using floats or float types variables. The format is as follows:



Example Number				
0	00	00	00	00
1	7F	00	00	00
-1	7F	80	00	00
10	82	20	00	00
100	85	48	00	00
123.45	85	76	E6	66
123.45E20	C8	27	4E	53
123.45 E-20	43	36	2E	17



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 does the compiler show less RAM than there really is?

Some devices make part of the RAM much more ineffective to access than the standard RAM. In particular, the 509, 57, 66, 67,76 and 77 devices have this problem.

By default, the compiler will not automatically allocate variables to the problem RAM and, therefore, the RAM available will show a number smaller than expected.

There are three ways to use this RAM:
1. Use #BYTE or #BIT to allocate a variable in this RAM. Do NOT create a pointer to these variables.

Example:

```
#BYTE counter=0x30
```

2. Use Read_Bank and Write_Bank to access the RAM like an array. This works well if needing to allocate an array in this RAM.

Example:

```
For(i=0;i<15;i++)
   Write_Bank(1,i,getc());
For(i=0;i<=15;i++)
   PUTC(Read_Bank(1,i));
```

3. Switch to larger pointers for full RAM access (which takes more ROM). In PCB add *=8 to the #device and in PCM/PCH add *=16 to the #device.

Example:

```
#DEVICE PIC16C77  *=16
```

or

```
#include <16C77.h>
#device *=16
```

Why does the compiler use the obsolete TRIS?

The use of TRIS causes concern for some users. The Microchip data sheets recommend not using TRIS instructions for upward compatibility. If there is existing ASM code and it uses TRIS, then it would be more difficult to port to a new Microchip part without TRIS. C does not have this problem, however; the compiler has a device database that indicates specific characteristics for every part. This includes information on whether the part has a TRIS and a list of known problems with the part. The latter question is answered by looking at the device errata.

CCS makes every attempt to add new devices and device revisions as the data and errata sheets become available.

PCW users can edit the device database. If the use of TRIS is a concern, simply change the database entry for the specific part and the compiler will not use it.

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. Use 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. Possibly 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, 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 the 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 BRGH1OK is specified in the #USE RS232 directive.

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 SIOW.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_14KAD.C
An analog to digital program with calibration for the PIC14000

EX_1920.C
Uses a Dallas DS1920 button to read temperature

EX_8PIN.C
Demonstrates the use of 8 pin PICs with their special I/O requirements

EX_92LCD.C
Uses a PIC16C92x chip to directly drive LCD glass

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_CCP1S.C
Generates a precision pulse using the PIC CCP module

EX_CCPMP.C
Uses the PIC CCP module to measure a pulse width

EX_COMP.C
Uses the analog comparator and voltage reference available on some PICs

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_DNSLOOKUP.C
Example to perform a DNS lookup on the internet

EX_DPOT.C
Controls an external digital POT

EX_DTMF.C
Generates DTMF tones

EX_EMAIL.C
Program will send e-mail

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_FLOAT.C
Shows how to use basic floating point

EX_FREQC.C
A 50 mhz frequency counter

EX_GLINT.C
Shows how to define a custom global interrupt hander for fast interrupts

EX_ICD.C
Shows a simple program for use with Microchips ICD debugger

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 for chips like the 16F877

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_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 will problem memory allocation

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_PBUTT.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 CCP module to generate a pulse stream

EX_REACT.C
Times the reaction time of a relay closing using the CCP module

EX_RMSDB.C
Calculates the RMS voltage and dB level of an AC signal

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_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_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_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 USB chip

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_WDT.C
Shows how to use the PIC watch dog timer

EX_WDT18.C
Shows how to use the PIC18 watch dog timer

EX_WEBSV.C
Shows how to implement a simple web server

EX_X10.C
Communicates with a TW523 unit to read and send power line X10 codes

LIST OF INCLUDE FILES (in the DRIVERS directory)

14KCAL.C

Calibration functions for the PIC14000 A/D converter

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

DNS.C
Functions used to perform a DNS lookup on the internet

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

PIC_USB.H
Hardware layer for built-in PIC USB

S7600.H
Driver for Sieko S7600 TCP/IP chip

SC28L19X.C
Driver for the Phillips external UART (4 or 8 port)

SETJMP.H
Standard C functions for doing jumps outside functions

SMTP.H
e-mail 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

```
////////////////////////////////////////////////////////////////
///                         EX_SQW.C                         ///
///   This program displays a message over the RS-232 and    ///
///   waits for any keypress to continue.  The program       ///
///   will then begin a 1khz square wave over I/O pin B0.    ///
///   Change both delay_us to delay_ms to make the           ///
///   frequency 1 hz.  This will be more visible on          ///
///   a LED.  Configure the CCS prototype card as follows:   ///
///   insert jumpers from 11 to 17, 12 to 18, and 42 to 47.  ///
////////////////////////////////////////////////////////////////

#ifdef __PCB__
#include <16C56.H>
#else
#include <16C84.H>
#endif

#use delay(clock=20000000)
#use rs232(baud=9600, xmit=PIN_A3, rcv=PIN_A2)
```

```
main()  {
   printf("Press any key to begin\n\r");
    getc();
    printf("1 khz signal activated\n\r");
    while (TRUE)  {
    output_high (PIN_B0);
    delay_us(500);
    output_low(PIN_B0);
    delay_us(500);
   }
}


////////////////////////////////////////////////////////////////
///                          EX_STWT.C                         ///
///    This program uses the RTCC (timer0) and interrupts     ///
///    to keep a real time seconds counter.  A simple stop    ///
///    watch function is then implemented.  Configure the     ///
///    CCS prototype card as follows, insert jumpers from:    ///
///    11 to 17 and 12 to 18.                                 ///
////////////////////////////////////////////////////////////////

#include <16C84.H>
#use delay (clock=20000000)
#use rs232(baud=9600, xmit=PIN_A3, rcv=PIN_A2_
#define INTS_PER_SECOND 76        //(20000000/(4*256*256))
byte seconds;                     //Number of interrupts left
                                  //before a second has elapsed

#int_rtcc                         //This function is called
clock_isr() {                     //every time the RTCC (timer0)
                                 //overflows (255->0)
                                  //For this program this is apx
                                 //76 times per second.

   if(--int_count==0)  {
   ++seconds;
   int_count=INTS_PER_SECOND;

   }
}

main() {
   byte start;
   int_count=INTS_PER_SECOND;
   set_rtcc(0);
   setup_counters (RTCC_INTERNAL, RTCC_DIV_256);
   enable_interrupts (INT_RTCC);
   enable_interrupts(GLOBAL)
   do {
       printf ("Press any key to begin. \n\r");
```

```
      getc();
      start=seconds;
      printf("Press any key to stop. \n\r");
      getc();
      printf ("%u seconds.  \n\r", seconds-start);
   } while (TRUE);
}


////////////////////////////////////////////////////////////////
///                          EX_INTEE.C                      ///
///    This program will read and write to the '83 or '84      ///
///    internal EEPROM.  Configure the CCS prototype card as  ///
///    follows: insert jumpers from 11 to 17 and 12 to 18.    ///
////////////////////////////////////////////////////////////////

#include <16C84.H>

#use delay(clock-100000000)
#use rs232 (baud=9600, xmit=PIN_A3, rv+PIN_A2)

#include <HEX.C>

main () {
   byte i,j,address, value;

   do {
       printf("\r\n\nEEPROM: \r\n")      //Displays contents
       for(i=0; i<3; ++i) {              //entire EEPROM
              for (j=0; j<=15; ++j) {    //in hex
                     printf("%2x", read_eeprom(i+16+j));

              }
              printf("\n\r");
       }
       printf ("\r\nlocation to change: ");
       address= gethex();
       printf ("\r\nNew value: ");
       value=gethex();

       write_eeprom (address, value);
   } while (TRUE)
}
```

```
////////////////////////////////////////////////////////////////
///     Library for a Microchip 93C56 configured for a x8      ///
///                                                            ///
///     org init_ext_eeprom();      Call before the other      ///
///                                 functions are used         ///
///                                                            ///
///     write_ext_eeprom(a,d);      Write the byte d to        ///
///                                 the address a              ///
///                                                            ///
///     d=read_ext_eeprom (a);      Read the byte d from       ///
///                                 the address a.             ///
///     The main program may define eeprom_select,             ///
///     eeprom_di, eeprom_do and eeprom_clk to override        ///
///     the defaults below.                                    ///
////////////////////////////////////////////////////////////////

#ifndef EEPROM_SELECT

#define EEPROM_SELECT      PIN_B7
#define EEPROM_CLK         PIN_B6
#define EEPROM_DI          PIN_B5
#define EEPROM_DO          PIN_B4

#endif

#define EEPROM_ADDRESS byte
#define EEPROM_SIZE        256

void init_ext_eeprom () {
   byte cmd[2];
   byte i;

   output_low(EEPROM_DI);
   output_low(EEPROM_CLK);
   output_low(EEPROM_SELECT);

   cmd[0]=0x80;
   cmd[1]=0x9;

   for (i=1; i<=4; ++i)
       shift_left(cmd, 2,0);
   output_high (EEPROM_SELECT);
   for (i=1; i<=12; ++i)  {
       output_bit(EEPROM_DI, shift_left(cmd, 2,0));
       output_high (EEPROM_CLK);
       output_low(EEPROM_CLK);
}
   output_low(EEPROM_DI);
   output_low(EEPROM_SELECT);
}
```

```
void write_ext_eeprom (EEPROM_ADDRESS address, byte data)  {
   byte cmd[3];
   byte i;

   cmd[0]=data;
   cmd[1]=address;
   cmd[2]=0xa;

   for(i=1;i<=4;++i)
       shift_left(cmd,3,0);
   output_high(EEPROM_SELECT);
   for(i=1;i<=20;++i) {
       output_bit (EEPROM_DI, shift_left (cmd,3,0));
       output_high (EEPROM_CLK);
       output_low(EEPROM_CLK);
   }
   output_low (EEPROM_DI);
   output_low (EEPROM_SELECT);
   delay_ms(11);
}

byte read_ext_eeprom(EEPROM_ADDRESS address) {
   byte cmd[3];
   byte i, data;

   cmd[0]=0;
   cmd[1]=address;
   cmd[2]=0xc;

   for(i=1;i<=4;++i)
       shift_left(cmd,3,0);
   output_high(EEPROM_SELECT);
   for(i=1;i<=20;++i) {
       output_bit (EEPROM_DI, shift_left (cmd,3,0));
       output_high (EEPROM_CLK);
       output_low(EEPROM_CLK);
       if (i>12)
              shift_left (&data, 1, input (EEPROM_DO));
   }
   output_low (EEPROM_SELECT);
   return(data);
}
```

```
////////////////////////////////////////////////////////////////
///   This file demonstrates how to use the real time         ///
///   operating system to schedule tasks and how to use       ///
///   the rtos_run function.                                  ///
///                                                           ///
///   this demo makes use of the PIC18F452 prototyping board  ///
////////////////////////////////////////////////////////////////

#include <18F452.h>
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
// this tells the compiler that the rtos functionality will be needed,
that
// timer0 will be used as the timing device, and that the minor cycle for
// all tasks will be 500 miliseconds
#use rtos(timer=0,minor_cycle=100ms)
// each function that is to be an operating system task must have the
#task
// preprocessor directive located above it.
// in this case, the task will run every second, its maximum time to run
is
// less than the minor cycle but this must be less than or equal to the
// minor cycle, and there is no need for a queue at this point, so no
// memory will be reserved.
#task(rate=1000ms,max=100ms)
// the function can be called anything that a standard function can be
called
void The_first_rtos_task ( )
{
   printf("1\n\r");
}
#task(rate=500ms,max=100ms)
void The_second_rtos_task ( )
{
   printf("\t2!\n\r");
}
#task(rate=100ms,max=100ms)
void The_third_rtos_task ( )
{
   printf("\t\t3\n\r");
}
// main is still the entry point for the program
void main ( )
{
   // rtos_run begins the loop which will call the task functions above at
the
   // schedualed time
   rtos_run ( );
}
```

```
////////////////////////////////////////////////////////////////
///   This file demonstrates how to use the real time         ///
///   operating system rtos_terminate function                ///
///                                                           ///
///   this demo makes use of the PIC18F452 prototyping board  ///
////////////////////////////////////////////////////////////////

#include <18F452.h>
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
#use rtos(timer=0,minor_cycle=100ms)
// a counter will be kept
int8 counter;
#task(rate=1000ms,max=100ms)
void The_first_rtos_task ( )
{
   printf("1\n\r");
   // if the counter has reached the desired value, the rtos will
terminate
   if(++counter==5)
      rtos_terminate ( );
}
#task(rate=500ms,max=100ms)
void The_second_rtos_task ( )
{
   printf("\t2!\n\r");
}
#task(rate=100ms,max=100ms)
void The_third_rtos_task ( )
{
   printf("\t\t3\n\r");
}
void main ( )
{
   // main is the best place to initialize resources the the rtos is
dependent
   // upon
   counter = 0;
   rtos_run ( );
   // once the rtos_terminate function has been called, rtos_run will
return
   // program control back to main
   printf("RTOS has been terminated\n\r");
}
```

```
////////////////////////////////////////////////////////////////
///   This file demonstrates how to use the real time         ///
///   operating system rtos_enable and rtos_disable functions ///
///                                                           ///
///   this demo makes use of the PIC18F452 prototyping board  ///
////////////////////////////////////////////////////////////////

#include <18F452.h>
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
#use rtos(timer=0,minor_cycle=100ms)
int8 counter;
// now that task names will be passed as parameters, it is best
// to declare function prototypes so that their are no undefined
// identifier errors from the compiler
#task(rate=1000ms,max=100ms)
void The_first_rtos_task ( );
#task(rate=500ms,max=100ms)
void The_second_rtos_task ( );
#task(rate=100ms,max=100ms)
void The_third_rtos_task ( );
void The_first_rtos_task ( ) {
   printf("1\n\r");
   if(counter==3)
   {
      // to disable a task, simply pass the task name
      // into the rtos_disable function
      rtos_disable(The_third_rtos_task);
   }
}
void The_second_rtos_task ( ) {
   printf("\t2!\n\r");
   if(++counter==10) {
      counter=0;
      // enabling tasks is similar to disabling them
      rtos_enable(The_third_rtos_task);
   }
}
void The_third_rtos_task ( ) {
   printf("\t\t3\n\r");
}
void main ( ) {
   counter = 0;
   rtos_run ( );
}
```

```
////////////////////////////////////////////////////////////////
///   This file demonstrates how to use the real time         ///
///   operating systems messaging functions                 ///
///                                                           ///
///   this demo makes use of the PIC18F452 prototyping board  ///
////////////////////////////////////////////////////////////////

#include <18F452.h>
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
#use rtos(timer=0,minor_cycle=100ms)
int8 count;
// each task will now be given a two byte queue
#task(rate=1000ms,max=100ms,queue=2)
void The_first_rtos_task ( );
#task(rate=500ms,max=100ms,queue=2)
void The_second_rtos_task ( );
void The_first_rtos_task ( ) {
   // the function rtos_msg_poll will return the number of messages in the
   // current tasks queue
   // always make sure to check that their is a message or else the read
   // function will hang
   if(rtos_msg_poll ( )>0){
      // the function rtos_msg_read, reads the first value in the queue
      printf("messages recieved by task1 : %i\n\r",rtos_msg_read ( ));
      // the funciton rtos_msg_send, sends the value given as the
      // second parameter to the function given as the first
      rtos_msg_send(The_second_rtos_task,count);
      count++;
   }
}
void The_second_rtos_task ( ) {
   rtos_msg_send(The_first_rtos_task,count);
   if(rtos_msg_poll ( )>0){
      printf("messages recieved by task2 : %i\n\r",rtos_msg_read ( ));
      count++;
   }
}
void main ( ) {
   count=0;
   rtos_run();
}
```

```
////////////////////////////////////////////////////////////////
///   This file demonstrates how to use the real time         ///
///   operating systems yield function                      ///
///                                                           ///
///   this demo makes use of the PIC18F452 prototyping board  ///
////////////////////////////////////////////////////////////////

#include <18F452.h>
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
#use rtos(timer=0,minor_cycle=100ms)
#task(rate=1000ms,max=100ms,queue=2)
void The_first_rtos_task ( );
#task(rate=500ms,max=100ms,queue=2)
void The_second_rtos_task ( );
void The_first_rtos_task ( ) {
   int count=0;
   // rtos_yield allows the user to break out of a task at a given point
   // and return to the same ponit when the task comes back into context
   while(TRUE){
      count++;
      rtos_msg_send(The_second_rtos_task,count);
      rtos_yield ( );
   }
}
void The_second_rtos_task ( ) {
   if(rtos_msg_poll( ))
   {
      printf("count is : %i\n\r",rtos_msg_read ( ));
   }
}
void main ( ) {
   rtos_run();
}


////////////////////////////////////////////////////////////////
///   This file demonstrates how to use the real time         ///
///   operating systems yield function signal and wait       ///
///   function to handle resources                           ///
///                                                           ///
///   this demo makes use of the PIC18F452 prototyping board  ///
////////////////////////////////////////////////////////////////

#include <18F452.h>
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
#use rtos(timer=0,minor_cycle=100ms)
// a semaphore is simply a shared system resource
// in the case of this example, the semaphore will be the red LED
int8 sem;
#define RED PIN_B5
```

```
#task(rate=1000ms,max=100ms,queue=2)
void The_first_rtos_task ( );
#task(rate=1000ms,max=100ms,queue=2)
void The_second_rtos_task ( );
void The_first_rtos_task ( ) {
   int i;
   // this will decrement the semaphore variable to zero which signals
   // that no more user may use the resource
   rtos_wait(sem);
   for(i=0;i<5;i++){
      output_low(RED); delay_ms(20); output_high(RED);
      rtos_yield ( );
   }
   // this will inrement the semaphore variable to zero which then signals
   // that the resource is available for use
   rtos_signal(sem);
}
void The_second_rtos_task ( ) {
   int i;
   rtos_wait(sem);
   for(i=0;i<5;i++){
      output_high(RED); delay_ms(20); output_low(RED);
      rtos_yield ( );
   }
   rtos_signal(sem);
}
void main ( ) {
   // sem is initialized to the number of users allowed by the resource
   // in the case of the LED and most other resources that limit is one
   sem=1;
   rtos_run();
}


////////////////////////////////////////////////////////////////
///   This file demonstrates how to use the real time         ///
///   operating systems await function                     ///
///                                                          ///
///   this demo makes use of the PIC18F452 prototyping board  ///
////////////////////////////////////////////////////////////////

#include <18F452.h>
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
#use rtos(timer=0,minor_cycle=100ms)
#define RED PIN_B5
#define GREEN PIN_A5
int8 count;
#task(rate=1000ms,max=100ms,queue=2)
void The_first_rtos_task ( );
#task(rate=1000ms,max=100ms,queue=2)
void The_second_rtos_task ( );
```

```
void The_first_rtos_task ( ) {
   // rtos_await simply waits for the given expression to be true
   // if it is not true, it acts like an rtos_yield and passes the system
   // to the next task
   rtos_await(count==10);
   output_low(GREEN); delay_ms(20); output_high(GREEN);
   count=0;
}
void The_second_rtos_task ( ) {
   output_low(RED); delay_ms(20); output_high(RED);
   count++;
}
void main ( ) {
   count=0;
   rtos_run();
}


////////////////////////////////////////////////////////////////
///   This file demonstrates how to use the real time         ///
///   operating systems statistics features                 ///
///                                                           ///
///   this demo makes use of the PIC18F452 prototyping board  ///
////////////////////////////////////////////////////////////////

#include <18F452.h>
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
#use rtos(timer=0,minor_cycle=100ms,statistics)
// This structure must be defined inorder to retrieve the statistical
// information
struct rtos_stats {
   int32 task_total_ticks;       // number of ticks the task has used
   int16 task_min_ticks;         // the minimum number of ticks used
   int16 task_max_ticks;         // the maximum number of ticks ueed
   int16 hns_per_tick;           // us = (ticks*hns_per_tic)/10
};
#task(rate=1000ms,max=100ms)
void The_first_rtos_task ( );
#task(rate=1000ms,max=100ms)
void The_second_rtos_task ( );
void The_first_rtos_task ( ) {
   struct rtos_stats stats;
   rtos_stats(The_second_rtos_task,&stats);
   printf ( "\n\r" );
   printf ( "task_total_ticks : %Lius\n\r" ,
            (int32)(stats.task_total_ticks)*stats.hns_per_tick );
   printf ( "task_min_ticks   : %Lius\n\r" ,
            (int32)(stats.task_min_ticks)*stats.hns_per_tick );
   printf ( "task_max_ticks   : %Lius\n\r" ,
            (int32)(stats.task_max_ticks)*stats.hns_per_tick );
   printf ("\n\r");
```

```
}
void The_second_rtos_task ( ) {
   int i, count = 0;
   while(TRUE) {
      if(rtos_overrun(the_second_rtos_task)) {
         printf("The Second Task has Overrun\n\r\n\r");
         count=0;
      }
      else
        count++;

      for(i=0;i<count;i++)
         delay_ms(50);

      rtos_yield();
   }
}
void main ( ) {
   rtos_run ( );
}


////////////////////////////////////////////////////////////////
///   This file demonstrates how to create a basic command    ///
///   line using the serial port withought having to stop     ///
///   RTOS operation, this can also be considered a           ///
///   semi kernal for the RTOS.                               ///
///                                                           ///
///   this demo makes use of the PIC18F452 prototyping board  ///
////////////////////////////////////////////////////////////////

#include <18F452.h>
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
#use rtos(timer=0,minor_cycle=100ms)
#define RED PIN_B5
#define GREEN PIN_A5
#include <string.h>
// this character array will be used to take input from the prompt
char input [ 30 ];
// this will hold the current position in the array
int index;
// this will signal to the kernal that input is ready to be processed
int1 input_ready;
// different commands
char en1 [ ] = "enable1";
char en2 [ ] = "enable2";
char dis1 [ ] = "disable1";
char dis2 [ ] = "disable2";
#task(rate=1000ms,max=100ms)
void The_first_rtos_task ( );
#task(rate=1000ms,max=100ms)
```

```
void The_second_rtos_task ( );
#task(rate=500ms,max=100ms)
void The_kernal ( );
// serial interupt
#int_rda
void serial_interrupt ( )
{
   if(index<29) {
      input [ index ] = getc ( );   // get the value in the serial recieve
reg
      putc ( input [ index ] );     // display it on the screen
      if(input[index]==0x0d){       // if the input was enter
         putc('\n');
         input [ index ] = '\0';    // add the null character
         input_ready=TRUE;          // set the input read variable to true
         index=0;                   // and reset the index
      }
      else if (input[index]==0x08){
         if ( index > 1 ) {
            putc(' ');
            putc(0x08);
            index-=2;
         }
      }
      index++;
   }
   else {
      putc ( '\n' );
      putc ( '\r' );
      input [ index ] = '\0';
      index = 0;
      input_ready = TRUE;
   }
}
void The_first_rtos_task ( ) {
   output_low(RED); delay_ms(50); output_high(RED);
}
void The_second_rtos_task ( ) {
   output_low(GREEN); delay_ms(20); output_high(GREEN);
}
void The_kernal ( ) {
   while ( TRUE ) {
      printf ( "INPUT:> " );
      while(!input_ready)
         rtos_yield ( );
      printf ( "%S\n\r%S\n\r", input , en1 );
      if ( !strcmp( input , en1 ) )
         rtos_enable ( The_first_rtos_task );
      else if ( !strcmp( input , en2 ) )
         rtos_enable ( The_second_rtos_task );
      else if ( !strcmp( input , dis1 ) )
```

```
            rtos_disable ( The_first_rtos_task );
         else if ( !strcmp ( input , dis2 ) )
            rtos_disable ( The_second_rtos_task );
         else
            printf ( "Error: unknown command\n\r" );
         input_ready=FALSE;
         index=0;
      }
   }
   void main ( ) {
      // initialize input variables
      index=0;
      input_ready=FALSE;
      // initialize interrupts
      enable_interrupts(int_rda);
      enable_interrupts(global);
      rtos_run();
   }
```

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
© 1994-2007 Custom Computer Services Incorporated
All Rights Reserved Worldwide
P.O. Box 2452
Brookfield, WI 53008