C Compiler
Reference Manual

Custom Computer Services Inc.
February 2002

Table of Contents

OVERVIEW 1

PCB, PCM AND PCH OVERVIEW 1
PCW OVERVIEW 1
TECHNICAL SUPPORT 1
INSTALLATION 2
INVOKING THE COMMAND LINE COMPILER 2
MPLAB INTEGRATION 3
DIRECTORIES 4
FILE FORMATS 4
DIRECT DEVICE PROGRAMMING 4
DEVICE CALIBRATION DATA 5
UTILITY PROGRAMS 5

PCW IDE 7

FILE MENU 7
PROJECT MENU 8
EDIT MENU 9
OPTIONS MENU 9
COMPILE OPTIONS 11
VIEW MENU 11
TOOLS MENU 13
HELP MENU 15
PCW EDITOR KEYS 16
PROJECT WIZARD 18

PRE-PROCESSOR 19

PRE-PROCESSOR DIRECTIVES 20
#ASM 20
#ENDASM 20
#BIT 23
#BYTE 24

#CASE.....25
_ _ DATE_ _.....25
#DEFINE.....26
#DEVICE.....27
_ _DEVICE_ _.....28
#ERROR.....28
_ _FILE_ _.....29
#FUSES.....29
#ID.....30
#IF EXPR.....31
#ELSE.....31
#ELIF.....31
#ENDIF.....31
#IFDEF.....32
#IFNDEF.....32
#ELSE.....32
#ELIF.....32
#ENDIF.....32
#INCLUDE.....33
#INLINE.....33
#INT_XXXX.....34
#INT_DEFAULT.....35
#INT_GLOBAL.....36
_ _LINE_ _.....36
#LIST.....37
#LOCATE.....37
#NOLIST.....37
#OPT.....38
#ORG.....38
_ _PCB_ _.....40
_ _PCM_ _.....40
_ _ PCH _ _.....41
#PRAGMA.....41
#PRIORITY.....41
#RESERVE.....42
#ROM.....42
#SEPARATE.....43
_ _ TIME _ _.....43
#TYPE.....44
#UNDEF.....45
#USE DELAY.....45
#USE FAST_IO.....46
#USE FIXED_IO.....46

#USE I2C 47
#USE RS232..47
#USE STANDARD_IO 49
#ZERO_RAM 50

DATA DEFINITIONS...51

FUNCTION DEFINITION 54

FUNCTION DEFINITION 54
REFERENCE PARAMETERS...55

C STATEMENTS AND EXPRESSIONS...56

PROGRAM SYNTAX 56
COMMENT 56
TRIGRAPH SEQUENCES...56
STATEMENTS 57
EXPRESSIONS 58
OPERATORS..59
OPERATOR PRECEDENCE 60

BUILT-IN FUNCTIONS 61

ABS( )..64
ACOS( )..64
ASIN( )..64
ASSERT()..64
ATAN( ) 65
ATAN2() 65
ATOF() 65
ATOI( )..66
ATOL( )..66
ATOI32()...66
BIT_CLEAR( ) 67
BIT_SET( ) 67
BIT_TEST( )...68
CEIL( )..69

COS( ) 69
COSH() 69
DELAY_CYCLES( ) 70
DELAY_MS( ) 70
DELAY_US( ) 71
DISABLE_INTERRUPTS( ) 72
ENABLE_INTERRUPTS( ) 73
EXP( ) 73
EXT_INT_EDGE( ) 74
FABS() 75
FLOOR( ) 75
FMOD() 76
FREXP() 77
GET_TIMERx() 77
GETC() 78
GETCH() 78
GETCHAR() 78
FGETC() 78
GETS( ) 79
FGETS() 79
GOTO_ADDRESS() 80
I2C_POLL( ) 81
I2C_READ() 81
I2C_START() 82
I2C_STOP() 83
I2C_WRITE( ) 83
INPUT( ) 84
INPUT_x() 85
ISAMOUNG( ) 86
ISALNUM(CHAR) 86
ISALPHA(char) 86
ISDIGIT(char) 86
ISLOWER(char) 86
ISSPACE(char) 86
ISUPPER(char) 86
ISXDIGIT(char) 86
ISCNTRL(x) 86
ISGRAPH(x) 86
ISPRINT(x) 86
ISPUNCT(x) 86
KBHIT() 88
LABEL_ADDRESS() 89
LABS( ) 89

LCD_LOAD( ) 90
LCD_SYMBOL( ) 90
LDEXP() 91
LOG( ) 92
LOG10( ) 93
MAKE8() 93
MAKE16() 94
MAKE32() 95
MEMCPY( ) 95
MEMSET( ) 96
MODF() 97
OFFSETOF() 97
OFFSETOFBIT() 97
OUTPUT_BIT( ) 98
OUTPUT_FLOAT( ) 99
OUTPUT_HIGH( ) 100
OUTPUT_LOW( ) 100
OUTPUT_A( ) 101
OUTPUT_B( ) 101
OUTPUT_C( ) 101
OUTPUT_D( ) 101
OUTPUT_E( ) 101
PERROR() 102
PORT_B_PULLUPS( ) 102
POW( ) 103
PRINTF( ) 103
FPRINTF() 103
PSP_OUTPUT_FULL() 105
PSP_INPUT_FULL() 105
PSP_OVERFLOW() 105
PUTC( ) 106
PUTCHAR( ) 106
FPUTC() 106
PUTS( ) 107
FPUTS() 107
READ_ADC() 107
READ_BANK( ) 108
READ_CALIBRATION( ) 109
READ_EEPROM( ) 110
READ_PROGRAM_EEPROM ( ) 110
RESET_CPU() 111
RESTART_CAUSE() 111
RESTART_WDT() 112

ROTATE_LEFT( ) 113
ROTATE_RIGHT( ) 114
SET_ADC_CHANNEL( ) 114
SET_PWM1_DUTY( ) 115
SET_PWM2_DUTY( ) 115
SET_RTCC( ) 116
SET_TIMER0( ) 116
SET_TIMER1( ) 116
SET_TIMER2( ) 116
SET_TIMER3( ) 116
SET_TRIS_A( ) 117
SET_TRIS_B( ) 117
SET_TRIS_C( ) 117
SET_TRIS_D( ) 117
SET_TRIS_E( ) 117
SET_UART_SPEED() 118
SETUP_ADC(MODE) 119
SETUP_ADC_PORTS( ) 119
SETUP_CCP1( ) 120
SETUP_CCP2( ) 120
SETUP_COMPARATOR( ) 121
SETUP_COUNTERS( ) 122
SETUP_LCD( ) 123
SETUP_PSP( ) 124
SETUP_SPI( ) 125
SETUP_TIMER_0 ( ) 125
SETUP_TIMER_1( ) 126
SETUP_TIMER_2( ) 127
SETUP_TIMER_3( ) 128
SETUP_VREF( ) 128
SETUP_WDT ( ) 129
SHIFT_LEFT( ) 130
SHIFT_RIGHT( ) 131
SIN ( ) 132
COS() 132
TAN() 132
ASIN() 132
ACOS() 132
ATAN() 132
SINH() 132
COSH() 132
TANH() 132
ATAN2() 132

SINH() 134
SLEEP() 134
SPI_DATA_IS_IN() 134
SPI_READ() 135
SPI_WRITE( ) 136
SPRINTF() 136
SQRT( ) 137
STANDARD STRING FUNCTIONS 138
STRCAT() 138
STRCHR() 138
STRRCHR() 138
STRCMP() 138
STRNCMP() 138
STRICMP() 138
STRNCPY() 138
STRCSPN() 138
STRSPN() 138
STRLEN() 138
STRLWR() 138
STRPBRK() 138
STRSTR() 138
STRTOK() 139
STRCPY( ) 140
SWAP( ) 141
TAN( ) 142
TANH() 142
TOLOWER( ) 142
TOUPPER( ) 142
WRITE_BANK( ) 143
WRITE_EEPROM( ) 143
WRITE_PROGRAM_EEPROM ( ) 144

STANDARD C DEFINITIONS 146

LIMITS.H 146
FLOAT.H 146
STDDEF.H 148
SETJMP.H 148

COMPILER ERROR MESSAGES 149

COMMON QUESTIONS AND ANSWERS 160

HOW DOES ONE MAP A VARIABLE TO AN I/O PORT?..161
WHY DOES A PROGRAM WORK WITH STANDARD I/O BUT NOT WITH FAST I/O? 163
WHY DOES THE GENERATED CODE THAT USES BIT VARIABLES LOOK SO UGLY? 164
WHY IS THE RS-232 NOT WORKING RIGHT? 165
HOW CAN I USE TWO OR MORE RS-232 PORTS ON ONE PIC®? 167
HOW DOES THE PIC® CONNECT TO A PC? 169
WHY DO I GET AN OUT OF ROM ERROR WHEN THERE SEEMS TO BE ROM LEFT?..170
WHAT CAN BE DONE ABOUT AN OUT OF RAM ERROR?..171
WHY DOES THE .LST FILE LOOK OUT OF ORDER?..172
HOW IS THE TIMER0 INTERRUPT USED TO PERFORM AN EVENT AT SOME RATE? 173
HOW DOES THE COMPILER HANDLE CONVERTING BETWEEN BYTES AND WORDS? 174
HOW DOES THE COMPILER DETERMINE TRUE AND FALSE ON EXPRESSIONS? 175
WHAT ARE THE RESTRICTIONS ON FUNCTION CALLS FROM AN INTERRUPT FUNCTION? 176
WHY DOES THE COMPILER USE THE OBSOLETE TRIS?...178
HOW DOES THE PIC® CONNECT TO AN I2C DEVICE? 179
INSTEAD OF 800, THE COMPILER CALLS 0. WHY? 180
INSTEAD OF A0, THE COMPILER IS USING REGISTER 20. WHY? 180
HOW DO I DIRECTLY READ/WRITE TO INTERNAL REGISTERS?...................................181
HOW CAN A CONSTANT DATA TABLE BE PLACED IN ROM?.......................................182
HOW CAN THE RB INTERRUPT BE USED TO DETECT A BUTTON PRESS? 183
WHAT IS THE FORMAT OF FLOATING POINT NUMBERS? 184
WHY DOES THE COMPILER SHOW LESS RAM THAN THERE REALLY IS?.....................185
WHAT IS AN EASY WAY FOR TWO OR MORE PICS® TO COMMUNICATE?....................186
HOW DO I WRITE VARIABLES TO EEPROM THAT ARE NOT A BYTE?.........................187
HOW DO I GET GETC() TO TIMEOUT AFTER A SPECIFIED TIME?................................188
HOW CAN I PASS A VARIABLE TO FUNCTIONS LIKE OUTPUT_HIGH()?....................189
HOW DO I PUT A NOP AT LOCATION 0 FOR THE ICD?...190
HOW DO I DO A PRINTF TO A STRING? 190
HOW DO I MAKE A POINTER TO A FUNCTION?..191
HOW MUCH TIME DOES MATH OPERATIONS TAKE? 192
HOW ARE TYPE CONVERSIONS HANDLED?...193

EXAMPLE PROGRAMS 195

SOFTWARE LICENSE AGREEMENT 208

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 the 16 and 18 bit PICmicro® MCU. Since much is in common between the compilers both are covered in this reference manual. Features and limitations that apply to only specific controllers are indicated within. These compilers are specially designed to meet the special needs of the PICmicro® MCU controllers. These tools allow developers to quickly design application software for these controllers in a highly readable high-level language.

The compilers have some limitations when compared to a more traditional C compiler. The hardware limitations make many traditional C compilers ineffective. As an example of the limitations, the compilers will not permit pointers to constant arrays. This is due to the separate code/data segments in the PICmicro® MCU hardware and the inability to treat ROM areas as data. On the other hand, the compilers have knowledge about the hardware limitations and does the work of deciding how to best implement your algorithms. The compilers can implement very efficiently normal C constructs, as well as input/output operations and bit twiddling operations.

PCW Overview

PCW is the professional package that includes both the PCM and PCB compilers. PCW has a Windows IDE. PCW has the same syntax as the command line compilers. The PCH compiler is available for PCW as an optional add-on.

Technical Support

The latest software can be downloaded via the Internet at:

http://www.ccsinfo.com/download.shtml

for 30 days after the initial purchase. For one year's worth of updates, you can purchase a Maintenance Plan directly from CCS. Also found on our web page are known bugs, the latest version of the software, and other news about the compiler.

We strive to ensure that each upgrade provides greater ease of use along with minimal, if any, problems. However, this is not always possible. To ensure that all problems that you encounter are corrected in a diligent manner, we suggest that you email us at support@ccsinfo.com outlining your specific problem along with an attachment of your file. This will ensure that solutions can be suggested to correct any problem(s) that may arise. We try to respond in a timely manner and take pride in our technical support.

Secondly, if we are unable to solve your problem by email, feel free to telephone us at (262) 797-0455 x 32. Please have all your supporting documentation on-hand so that your questions can be answered in an efficient manner. Again, we will make every attempt to solve any problem(s) that you may have. Suggestions for improving our software are always welcome and appreciated.

Installation

PCB, PCM, and PCH Installation:
Insert the disk in drive A and from Windows Start|Run type:

A:SETUP

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

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
+F7	Select PC7 (PIC17XXX)	+DC	Expanded .COD format debug file
+FS	Select PCS (SX)	+Yx	Optimization level x (0-9)
+ES	Standard error file	+T	Create call tree (.TRE)
+EO	Old error file format	+A	Create stats file (.STA)
-J	Do not create PJT file	-M	Do not create symbol 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 and debug files on disk after compile
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.
#xxx="yyy"	Set a global #define for id xxx with a value of yyy, example: **`#debug="true"`**
+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)

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
```

MPLAB Integration

The CCSC.EXE Windows program will work as a bridge from MPLAB to the C compiler. Simply enter the following from Start|Run type:

CCSC +SETUP

This will configure MPLAB. When creating a new project select CCS as the LANGUAGE TOOL SUITE. Then select the .HEX file and click on NODE

PROPERTIES. Here you need to select the compiler you want to use (PCB, PCM, and PCH).

If your first compile is done from the CCS IDE then it will create a MPLAB project file eliminating the need to create a new project and edit the nodes as described above.

If your MPLAB version is older than 3.40, you will need to download the latest version from Microchip's web page at: http://www.Microchip.com

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.

File Formats

The compiler can output 8 bit hex, 16 bit hex, and binary files. Two listing formats are available. Standard format resembles the Microchip tools and may be required by some third-party tools. The simple format is easier to read. The debug file may either be a Microchip .COD file or Advanced Transdata .MAP file. All file formats and extensions are selected via the **options|file** formats window on the DOS IDE and the **compiler|options** in the Windows IDE.

Direct Device Programming

The IDE has a program option in the main menu bar. When invoked, the IDE will issue a command to start the user's device programmer. The commands are specified in the **Options|Programer Options** window. The %H is replaced with the HEX filename and %D is replaced with the device number. Put a ! at the end if the command line if you would like a pause before returning to IDE. Only programs that can be invoked by a command will work with this option.

Device Calibration Data

Some devices from Microchip have calibration data programmed into the program area when shipped from the factory. Each part has its own unique data. This poses some special problems during development. When an UV erasable (windowed) part is erased, the calibration data is erased as well. Calibration data can be forced into the chip during programming by using a #ROM directive with the appropriate data.

The PCW package includes a utility program to help streamline this process. When a new chip is purchased, the chip should be read into a hex file. Execute the File|Read calibration data utility and select a name (.C) for this part. The utility will create an Include File with specified name that will have the correct #ROM directives for the part. During prototype development add a #Include directive and change the name before each build to the part # that is about to be programmed. For production (OTP parts) simply comment out the #Include.

Utility Programs

SIOW
SIOW is a Windows utility (PCW only). SIOW is a simple "dumb terminal" program that may be run on a PC to perform input and output over a serial port. SIO is handy since it will show all incoming characters. If the character is not a normally displayable character, it will show the hex code.

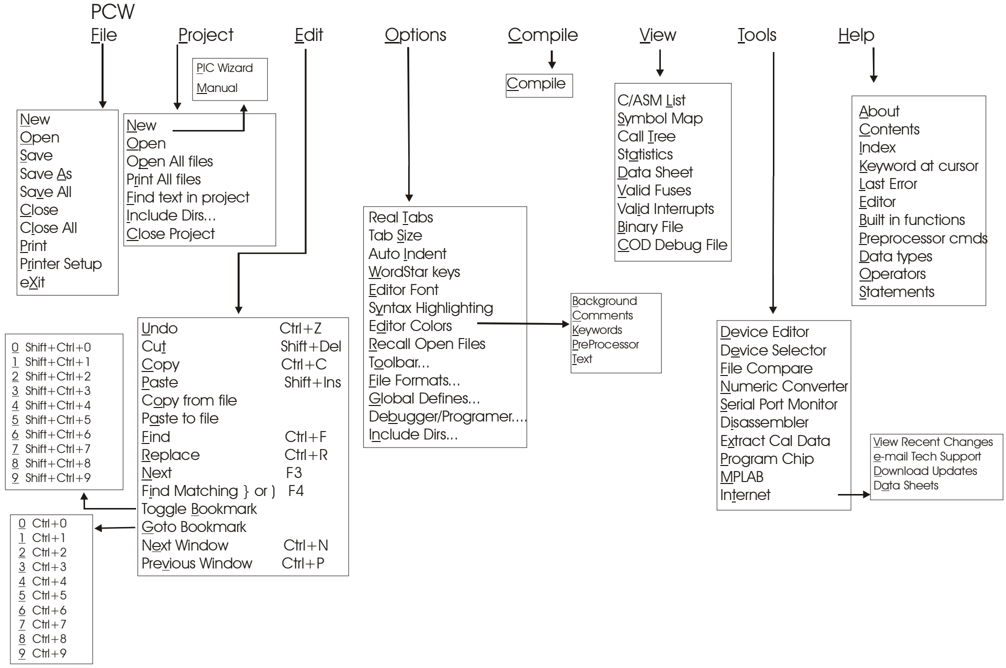
DEVEDIT
DEVEDIT is a Windows utility (PCW only) that will edit the device database. The compiler uses the device database to determine specific device characteristics at compile time. This utility will allow devices to be added, modified or removed. To add a device, highlight the closest equivalent chip and click on ADD. To edit or delete, highlight the device and click on the appropriate button.

PCONVERT
PCONVERT is a Windows utility (PCW only) that will perform conversions from various data types to other types. For example, Floating Point decimal to 4 BYTE Hex. The utility opens a small window to perform the conversions. This window can remain active during a PCW or MPLAB session. This can be useful during debugging.

CCSC +Q
This will list all devices in the compiler database.

CCSC +FM +V
This will show the current compiler version. Replace +FM with +FB or +FH for the other compilers.

PCW IDE



File Menu

New	Creates a new file
Open	Opens a file into the editor. If there are no other files open then the project name is set to this files name. Ctrl-O is the shortcut.
Save	Saves the file currently selected for editing. Ctrl-S is the shortcut.
Save As	Prompts for a filename to save the currently selected file.
Save All	All open files are saved to disk
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.
Print	Prints the currently selected file.
Printer Setup	Allows the selection of a printer and the printer settings.
Exit	Terminates PCW

Project Menu

New	Creates a new project. A project may be created manually or via a wizard. If created manually only a .PJT file is created to hold basic project information. An existing .C main file may be specified or an empty one may be created. The wizard will allow the user to specify project parameters and when complete a .C, .H and .PJT file are created. Standard source code and constants are generated based on the specified project parameters.
NEW PROJECT (Speed button or File|New Project)	This command will bring up a number of fill-in-the-blank forms about your new project. RS232 I/O and 12C characteristics, timer options, interrupts used, A/D options, drivers needed and pin names all may be specified in the forms. When drivers are selected, required pins will be selected by the tool and pins that can be combined will be. Final pins selections may be edited by the user. After all selections are made the initial .c and .h files are created with #defines, #includes and initialization commands required for your project. This is a fast way to start a new project. Once the files are created you cannot return to the menus to make further changes.
Open	A .PJT file is specified and the main source file is loaded.
Open All Files	A .PJT file is specified and all files used in the project are opened. In order for this function to work the program must have been compiled in order for the include files to become known.
Find Text In Project	Searches all files in a project for a given text string.
Print All Files	All files in the project are printed. In order for this function to work the program must have been compiled in order for the include files to become known.
Include Dirs	Allows the specification of each directory to be used to search for include files for just this project. This information is saved in the .PJT file.
Close Project	Closes all files associated with the current project.

Edit Menu

Undo	Undoes the last deletion.
Cut	Moves the selected text from the file to the clipboard.
Copy	Copies the selected text to the clipboard.
Paste	Copies the clipboard contents to the cursor location.
Copy from File	Copies the contents of a file to the cursor location.
Paste to File	Pastes the selected text to a file.
Find	Searches for a specified string in the file.
Replace	Replaces a specified string with a new string.
Next	Performs another Find or Replace.
Find matching } or)	The text will be highlighted up to the corresponding } or). The editor will start counting the open and close curly braces and highlight the closing item when they are balanced. Simply place the cursor before or on the element you need to find a match for and click, and the match will be highlighted.
Toggle Bookmark	Sets a bookmark (0-9) at the cursor location.
Goto Bookmark	Move the cursor to the specified bookmark (0-9).
Next Window	Selects the next open file as the current file for editing.
Previous Window	Selects the previous open file as the current file for editing.

Options Menu

Real tabs	When selected the editor inserts a tab character (ASCII 9) when the TAB key is pressed. When it is not selected and the TAB key is pressed spaces are inserted up to the next tab position.

Tab size	Determines the number of characters between tab positions. Tabs allow you to set the number of space equated by a tab and whether or not the tabs are converted to spaces or left as tabs.
Auto indent	When selected and the ENTER is pressed the cursor moves to the next line under the first character in the previous line. When not selected the ENTER always moves to the beginning of the next line.
WordStar keys	When selected the editing keys are WordStar style. WordStar commands will enable additional keystrokes recognized by the editors. See EDITOR for more information.
Syntax Highlighting	When checked the editor highlights in color C keywords and comments.
Toolbar	Allows the selection of what menu items appear as buttons on the toolbar.
Editor Font	Selects the editor font.
Editor Colors	Selects the colors used for syntax highlighting.
Recall Open Files	When selected PCW will always start with the same files open as were open when it last shut down. When not selected PCW always starts with no files open.
File Formats	Allows selection of the output file formats,
Programmer options	Allows the specification of the device programmer to be used when the PROGRAM CHIP tool is selected.
Include Dirs	Allows the specification of each directory to be used to search for include files by default for newly created projects. This has no effect on projects already created (use Project|Include Dirs to change those).
Global Definitions	Allows the setting of #defines to be used in compiling. This is the same as having some #defines at the top of your program. This may be used for example to set debugging defines without changing the code.

Compile Options

PCB/PCM (speed button or compile|PCx)
This command will compile your program. Use PCB for the 12-bit chips and PCM for the 14-bit chips.

PCW Compile
Compiles the current project (name is in lower right) using the current compiler (name is on the toolbar).

Debug File Options	
Microchip COD	Standard PICmicro® MCU debug file
RICE16 MAP	Used only be older RICE16 S/W
To Extended COD	COD file with advanced debug info

List Format Options	
Simple	A basic format with C code and ASM
Standard	The MPASM standard format with machine code
Old	Older MPASM format
Object file extension	The file extension for a HEX file
List file extension	The file extension for a list file
Symbolic	Includes C symbols within the assembly

Object File Options	
8 bit HEX	8 Bit Intel HEX file
16 bit HEX	16 bit Intel HEX file
Binary	Straight binary (No fuse info)

Error File Options	
Standard	Current Microchip standard
Original	Older Microchip standard

View Menu

C/ASM	Opens the listing file in the read only mode. The file must have been compiled to view the list file. If open this file will be updated after each compile. The listing file shows each C source line and the associated assembly code generated for the line.

For Example:

```
..............delay_ms(3);
0F2:   MOVLW 05
0F3:   MOVWF 08
0F4:   DESCZ 08,F
0F5:   GOTO 0F4
...............while input(pin_0));
0F6:   BSF 0B,3
```

Symbol Map

Opens the symbol file in the read only mode. The file must have been compiled to view the symbol file. If open this file will be updated after each compile. The symbol map shows each register location and what program variables are saved in each location.

MAP

Displays the RAM memory map for the program last compiled. The map indicates the usage of each RAM location. Some locations have multiple definitions since RAM is reused depending on the current procedure being executed.

For Example:

```
08     @SCRATCH
09     @SCRATCH
0A     TRIS_A
0B     TRIS_B
0C     MAIN.SCALE
0D     MAIN.TIME
0E     GET_SCALE.SCALE
0E     PUTHEX.N
0E     MAIN.@SCRATCH
```

Call Tree

Opens the tree file in the read only mode. The file must have been compiled to view the tree file. If open this file will be updated after each compile. The call tree shows each function and what functions it calls along with the ROM and RAM usage for each.

A (inline) will appear after inline procedures that begin with @. After the procedure name is a number of the form s/n where s is the page number of the procedure and n is the number is locations of code storage is required. If S is ? then this was the last procedure attempted when the compiler ran out of ROM space. RAM=xx indicates the total RAM required for the function.

For Example:

```
Main 0/30
        INIT 0/6
        WAIT_FOR_HOST 0/23 (Inline)
                DELAY_US 0/12
        SEND_DATA 0/65
```

Statistics — Opens the stats file in the read only mode. The file must have been compiled to view the stats file. If open this file will be updated after each compile. The statistics file shows each function, the ROM and RAM usage by file, segment and name.

Data Sheet — This tool will bring up Acrobat Reader with the manufacture data sheet for the selected part. If data sheets were not copied to disk then the CCS CD ROM or a manufacture CD ROM must be inserted.

Binary file — Opens a binary file in the read only mode. The file is shown in HEX and ASCII.

COD Debug file — Opens a debug file in the read only mode. The file is shown in an interpreted form.

Valid Fuses — Shows a list of all valid keywords for the #fuses directive for this device.

Valid Interrupts — Shows a list of all valid keywords for the #int_xxxx directive and enable/disable _interrupts for this device.

Status Line — Click on the left hand side of the status line to GOTO a specific line number.

Tools Menu

Device Editor — This tool allows the essential characteristics for each supported processor to be specified. This tool edits a database used by the compiler to control the compilation. CCS maintains this database (Devices.dat) however users may want to add new devices or change the entries for a device for a special application. Be aware if the database is changed and then the software is updated the changes will be lost. Save your DEVICES.DAT file during an update to prevent this.

Device selector
This tool uses the device database to allow a parametric selection of devices. By selecting key characteristics the tool displays all eligible devices.

File Compare
Compares two files. When source file is selected then a normal line by line compare is done. When list file is selected the compare may be set to ignore RAM and/or ROM addresses to make the comparison more meaningful. For example if an asm line was added at the beginning of the program a normal compare would flag every line as different. By ignoring ROM addresses then only the extra line is flagged as changed. Two output formats are available. One for display and one for files or printing.

Numeric Converter
A conversion tool to convert between decimal, hex and float.

Serial Port Monitor
An easy to use tool to connect to a serial port. This tool is convenient to communicate with a target program over an RS232 link. Data is shown in as ASCII characters and as raw hex.

Disassembler
This tool will take as input a HEX file and will output ASM. The ASM may be in a form that can be used as inline ASM.

This command will take a HEX file and generate an assembly file so that selected sections can be extracted and inserted into your C programs as inline assembly. Options will allow the selection of the assembly format.

- 12 or 14 bit opcodes
- Address, C, MC ASM labels
- Hex or Binary
- Simple, ASM, C numbers

Extract Cal Data
This tool will take as input a HEX file and will extract the calibration data to a C include file. This may be used to maintain calibration data for a UV erasable part. By

	including the include file in a program the calibration data will be restored after re-burning the part.
Program Chip	This simply invokes device programmer software with the output file as specified in the Compile\Options window. This command will invoke the device programmer software of your choice. Use the compile options to establish the command line.
MPLAB	Invokes MPLAB with the current project. The project is closed so MPLAB may modify the files if needed. When MPLAB is invoked this way PCW stays minimized until MPLAB terminates and then the project is reloaded.
Internet	These options invoke your WWW browser with the requested CCS Internet page:

• View recent changes	Shows version numbers and changes for the last couple of months.
• e-mail technical support	Starts your e-mail program with CCS technical support as the To: address.
• Download updates	Goes to the CCS download page. Be sure to have your reference number ready.
• Data Sheets	A list of various manufacture data sheets for devices CCS has device drivers for (such as EEPROMs, A/D converters, RTC...)

Help Menu

About	Shows the version of the IDE and each installed compiler.
Contents	The help file table of contents.
Index	The help file index.
Keyword at cursor	Does an index search for the keyword at the cursor location. Just press F1 to use this feature.
F12	Bring up help index
Shift F12	Bring up editor help

PCW Editor Keys

Cursor Movement	
Left Arrow	Move cursor one character to the left
Right Arrow	Move cursor one character to the right
Up Arrow	Move cursor one line up
Down Arrow	Move cursor one line down
Ctrl Left Arrow	Move cursor one word to the left
Ctrl Right Arrow	Move cursor one word to the right
Home	Move cursor to start of line
End	Move cursor to end of line
Ctrl PgUp	Move cursor to top of window
Ctrl PgDn	Move cursor to bottom of window
PgUp	Move cursor to previous page
PgDn	Move cursor to next page
Ctrl Home	Move cursor to beginning of file
Ctrl End	Move cursor to end of file
Ctrl S	Move cursor one character to the left
Ctrl D	Move cursor one character to the right
Ctrl E	Move cursor one line up
Ctrl X	** Move cursor one line down
Ctrl A	Move cursor one word to the left
Ctrl F	Move cursor one word to the right
Ctrl Q S	Move cursor to top of window
Ctrl Q D	Move cursor to bottom of window
Ctrl R	Move cursor to beginning of file
Ctrl C	* Move cursor to end of file
Shift ~	Where ~ is any of the above: Extend selected area as cursor moves

Editing Commands	
F4	Select next text with matching () or {}
Ctrl #	Goto bookmark # 0-9
Shift Ctrl #	Set bookmark # 0-9
Ctrl Q #	Goto bookmark # 0-9
Ctrl K #	Set bookmark # 0-9
Ctrl W	Scroll up
Ctrl Z	* Scroll down
Del	Delete the following character
BkSp	Delete the previous character
Shift BkSp	Delete the previous character
Ins	Toggle Insert/Overwrite mode
Ctrl Z	** Undo last operation
Shift Ctrl Z	Redo last undo
Alt BkSp	Restore to original contents
Ctrl Enter	Insert new line
Shift Del	Cut selected text from file
Ctrl Ins	Copy selected text
Shift Ins	Paste
Tab	Insert tab or spaces
Ctrl Tab	Insert tab or spaces
Ctrl P ~	Insert control character ~ in text
Ctrl G	Delete the following character
Ctrl T	Delete next word
Ctrl H	Delete the previous character
Ctrl Y	Delete line
Ctrl Q Y	Delete to end of line
Ctrl Q L	Restore to original contents
Ctrl X	** Cut selected text from file
Ctrl C	** Copy selected text
Ctrl V	Paste
Ctrl K R	Read file at cursor location
Ctrl K W	Write selected text to file
Ctrl-F	** Find text
Ctrl-R	** Replace text
F3	Repeat last find/replace

* Only when WordStar mode selected
** Only when WordStar mode is not selected

Project Wizard

The new project wizard makes starting a new project easier.

After starting the Wizard you are prompted for the name for your new main c file. This file will be created along with a corresponding .h file.

The tabbed notebook that is displayed allows the selection of various project parameters. For example:

- General Tab -> Select the device and clock speed
- Communications tab --> Select RS232 ports
- I/O Pins tab --> Select you own names for the various pins

When any tab is selected you may click on the blue square in the lower right and the wizard will show you what code is generated as a result of your selections in that screen.

After clicking OK all the code is generated and the files are opened in the PCW editor

This command will bring up a number of fill-in-the-blank forms about your new project. RS232 I/O and 12C characteristics, timer options, interrupts used, A/D options, drivers needed and pin names all may be specified in the forms. When drivers are selected, required pins will be selected by the tool and pins that can be combined will be. Final pins selections may be edited by the user. After all selections are made an initial .c and .h files are created with #defines, #includes and initialization commands require for your project. This is a fast way to start a new project. Once the files are created you cannot return to the menus to make further changes.

PRE-PROCESSOR

Pre-Processor Command Summary			
Standard C		**Device Specification**	
#DEFINE ID STRING	p.26	#DEVICE CHIP	p.27
#ELSE	p.31	#ID NUMBER	p.30
#ENDIF	p.31	#ID "filename"	p.30
#ERROR	p.28	#ID CHECKSUM	p.30
#IF expr	p.31	#FUSES options	p.29
#IFDEF id	p.32	#TYPE type=type	p.44
#INCLUDE "FILENAME"	p.33	**Built-in Libraries**	
#INCLUDE <FILENAME>	p.33	#USE DELAY CLOCK	p.45
#LIST	p.37	#USE FAST_IO	p.46
#NOLIST	p.37	#USE FIXED_IO	p.46
#PRAGMA cmd	p.41	#USE I2C	p.47
#UNDEF id	p.45	#USE RS232	p.47
Function Qualifier		#USE STANDARD_IO	p.49
#INLINE	p.33	**Memory Control**	
#INT_DEFAULT	p.35	#ASM	p.20
#INT_GLOBAL	p.36	#BIT id=const.const	p.23
#INT_xxx	p.34	#BIT id=id.const	p.22
#SEPARATE	p.43	#BYTE id=const	p.24
Pre-Defined Identifier		#BYTE id=id	p.22
_ _ DATE_ _	p.25	#LOCATE id=const	p.37
_ _ DEVICE_ _	p.28	#ENDASM	p.20
_ _ FILE_ _	p.29	#RESERVE	p.42
_ _LINE_ _	p.36	#ROM	p.42
_ _ PCB _ _	p.40	#ZERO_RAM	p.50
_ _ PCM_ _	p.40	**Compiler Control**	
_ _ PCH_ _	p.41	#CASE	p.25
_ _TIME_ _	p.43	#OPT n	p.38
		#PRIORITY	p.41
		#ORG	p.38

Pre-Processor Directives

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
```

#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
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	
	Only PCM:
	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,a	ADDWFC	f,d,a	ANDWF	f,d,a
CLRF	f,a	COMF	f,d,a	CPFSEQ	f,a
CPFSGT	f,a	CPFSLT	f,a	DECF	f,d,a
DECFSZ	f,d,a	DCFSNZ	f,d,a	INCF	f,d,a
INFSNZ	f,d,a	IORWF	f,d,a	MOVF	f,d,a
MOVFF	fs, fd (subscript s and d)	MOVWF	f,a	MULWF	f,a
NEGF	f,a	RLCF	f,d,a	RLNCF	f,d,a
RRCF	f,d,a	RRNCF	f,d,a	SETF	f,a
SUBFWB	f,d,a	SUBWF	f,d,a	SUBWFB	f,d,a
SWAPF	f,d,a	TSTFSZ	f,a	XORWF	f,d,a
BCF	f,b,a	BSF	f,b,a	BTFSC	f,b,a
BTFSS	f,b,a	BTG	f,d,a	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	+*				

#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 y and bit x. 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

#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 know 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.

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-02"

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, 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***

Elements: ***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.

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

Purpose: Defines the target processor. Every program must have exactly one #device with a chip.

Examples:

```
#device PIC16C74
#device PIC16C67 *=16
#device *=16  ICD=TRUE
#device PIC16F877 *=16 ADC=10
```

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_port_a( ALL_DIGITAL );
#endif
```

Example Files: None

Also See: #device

#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

_ _FILE_ _

Syntax: _ _FILE_ _

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: " __FILE__
          " at line " __LINE__ "\r\n");
```

Example Files: assert.h

Also see: _ _ line_ _

#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.

Examples:

```
#fuses  HS,NOWDT
```

Example Files: ex_sqw.c

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

#IF expr
#ELSE
#ELIF
#ENDIF

Syntax:

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

Elements:

expr is an expression with constants, standard operators and/or preprocessor identifiers. ***Code*** is any standard c source code.

Purpose:

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

Note: you may NOT use C variables in the #IF only preprocessor identifiers created via #define.

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

Examples:

```
#if MAX_VALUE > 255
long value;
#else
int value;
#endif
```

Example Files: ex_extee.c

Also See: #ifdef, #ifndef

#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 nay 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

#INCLUDE

Syntax: #include <***filename***>
or
#include "***filename***"

Elements: ***filename*** is a valid PC filename. It may include normal drive and path information.

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 s searched last.

Examples:

```
#include  <16C54.H>

#include <C:\INCLUDES\COMLIB\MYRS232.C>
```

Example Files: ex_sqw.c

Also See: None

#INLINE

Syntax: #inline

Elements: None

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

Examples:

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

```
b=t;
}
```

Example Files: ex_cust.c

Also See: #separate

#INT_xxxx

Syntax:

#INT_AD	Analog to digital conversion complete
#INT_ADOF	Analog to digital conversion timeout
#INT_BUSCOL	Bus collision
#INT_BUTTON	Pushbutton
#INT_CCP1	Capture or Compare on unit 1
#INT_CCP2	Capture or Compare on unit 2
#INT_COMP	Comparator detect
#INT_EEPROM	write complete
#INT_EXT	External interrupt
#INT_EXT1	External interrupt #1
#INT_EXT2	External interrupt #2
#INT_I2C	I2C interrupt (only on 14000)
#INT_LCD	activity
#INT_LOWVOLT	Low voltage detected
#INT_PSP	Parallel Slave Port data in
#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_RTCC	Timer 0 (RTCC) overflow
#INT_SSP	SPI or I2C activity
#INT_TBE	RS232 transmit buffer empty
#INT_TIMER0	Timer 0 (RTCC) overflow
#INT_TIMER1	Timer 1 overflow
#INT_TIMER2	Timer 2 overflow
#INT_TIMER3	Timer 3 overflow

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.

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

#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
#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

#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: ***n*** is the optimization level 0-9

Purpose: The optimization level is set with this directive. The directive 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,PCH stand-alone compilers. The PCW default is 9 for full optimization. This 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***

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.

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.

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.

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.

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 this directive does not prevent the ROM area from being used. See #ORG to reserve ROM.

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

_ _ 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***

Elements:

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 will 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.

Examples:

```
#TYPE  SHORT=8, INT=16, LONG=32
```

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***)
or
#use delay(***clock***=***speed***, ***restart_wdt***)

Elements: ***speed*** is a constant 1-100000000 (1 hz to 100 mhz)

Purpose: Tells the compiler the speed of the processor and enables the use of the built-in functions: delay_ms() and delay_us(). Speed is in cycles per second. An optional restart_WDT may be used to cause the compiler to restart the WDT while delaying.

Examples:

```
#use delay (clock=20000000)
#use delay (clock=32000, RESTART_WDT)
```

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().

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.

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:

- **MASTER**- Set the 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
- **SLOW**- Use the slow I2C specification
- **RESTART_WDT**- Restart the WDT while waiting in I2C_READ
- **FORCE_HW**-Use hardware I2C functions.

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.

Examples:

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

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

Example Files: ex_extee.c with 2464.c

Also See: i2c_read(), i2c_write()

#USE RS232

Syntax: #use rs232 (***options***)

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

• BAUD=x	Set baud rate to x
• XMIT=pin	Set transmit pin
• RCV=pin	Set receive 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 SCI.
• 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.
• ERRORS-	Used to cause the compiler to keep receive errors in the variable RS232_ERRORS and to reset errors when they occur.
• 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.
• STREAM=streamed-	Associates a stream identifier with this RS232 port. The identifier may then be used in functions like fputc.

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.

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.

Examples:

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

Example Files: ex_sqw.c

Also See: getc(), putc(), printf()

#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.

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

DATA DEFINITIONS

Data Types

The following tables show the syntax for data definitions. If the keyword TYPEDEF is used before the definition then the identifier does not allocate space but rather may be used as a type specifier in other data definitions. If the keyword CONST is used before the identifier, the identifier is treated as a constant. Constants must have an initializer and may not be changed at run-time. Pointers to constants are not permitted.

SHORT is a special type used to generate very efficient code for bit operations and I/O. Arrays of SHORT and pointers to SHORT are not permitted. Note: [ ] in the following tables indicates an optional item.

Data Declarations

```
[type-qualifier]      [type-specifier]      [declarator];
enum       [id]       { [ id [ = cexpr] }
                              ^
                              |
                      One or more comma separated

struct     [id]     {  [ type-qualifier [ [*]      id      cexpr [ cexpr ] ]]}
  or                         ^                                     ^
union                        |                                     |
                    One or more semi-             Zero or more
                    colon separated

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

Type Qualifer	
static	Variable is globally active and initialized to 0
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	Is allowed as a qualifier however, has no effect.
register	Is allowed as a qualifier however, has no effect.

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

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.

Type-id	An id from a TYPE definition
Enum	An enumerated type, syntax below
Struct	A structure, syntax in next table
Union	A union, syntax in next table

declarator

```
[const]          [*]       id          [ cexpr ]      [= init]
                           ^
                           |
                           Zero or more
```

enum:

```
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 =cepr follows an id that id will have the value of the constant expression and the following list will increment by one.

struct and union					
struct	[id]	{ [type-qualifier [[*]	id	cexpr [cexpr]]]}	
union		^		^	
		|		|	
		One or more semi-colon		Zero or more	

The :cexpr after an id specifies in a struct or union the number of bits to use for the id. This number may be 1-8. Multiple [] may be used for multiple dimension arrays. Structures and unions may be nested. The id after STRUCT may be used in another STRUCT and the {} is not used to reuse the same structure form again.

Examples:

```
int a,b,c,d;
typedef int byte;
typedef short bit;

bit e,f;
byte g[3][2];
char *h;
enum boolean  {false, true};
boolean j;
byte k = 5;
byte const   WEEKS = 52;
byte const FACTORS [4] =
    {8, 16, 64, 128};

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

FUNCTION DEFINITION

Function Definition

The format of a function definition is as follows:

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

The qualifiers for a function are as follows:

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

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

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

Example:

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

lcd_putc ("Hi There.");
```

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);
```

C STATEMENTS AND EXPRESSIONS

Program Syntax

A program is made up of the following four elements in a file. These are covered in more detail in the following paragraphs.

- Comment
- Pre-Processor Directive
- Data Definition
- Function Definition

Comment

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

Trigraph Sequences

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

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

STATEMENTS

STATEMENT	EXAMPLE

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

Constants:	
123	Decimal
0123	Octal
0x123	Hex
0b010010	Binary
'x'	Character
'\010'	Octal Character
'\xA5	Hex Character
'\c'	Special Character. Where c is one of: \n Line Feed- Same as \x0a \r Return Fee - Same as \x0d \t TAB- Same as \x09 \b Backspace- Same as \x08 \f Form Feed- Same as x0c \a Bell- Same as \x07 \v Vertical Space- Same as \x0b \? Question Mark- Same as \x3f \’ Single Quote- Same as \x60 \” Double Quote- Same as \x22 \\ A Single Backslash- Same as \x5c
"abcdef"	String (null is added to the end)

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

Operators

<table>
<tr><td>+</td><td>Addition Operator</td></tr>
<tr><td>+=</td><td>Addition assignment operator, x+=y, is the same as x=x+y</td></tr>
<tr><td>&=</td><td>Bitwise and assignment operator, x&=y, is the same as x=x&y</td></tr>
<tr><td>&</td><td>Address operator</td></tr>
<tr><td>&</td><td>Bitwise and operator</td></tr>
<tr><td>^=</td><td>Bitwise exclusive or assignment operator, x^=y, is the same as x=x^y</td></tr>
<tr><td>^</td><td>Bitwise exclusive or operator</td></tr>
<tr><td>|=</td><td>Bitwise inclusive or assignment operator, x|=y, is the same as x=x|y</td></tr>
<tr><td>|</td><td>Bitwise inclusive or operator</td></tr>
<tr><td>?:</td><td>Conditional Expression operator</td></tr>
<tr><td>- -</td><td>Decrement</td></tr>
<tr><td>/=</td><td>Division assignment operator, x\=y, is the same as x=x/y</td></tr>
<tr><td>/</td><td>Division operator</td></tr>
<tr><td>==</td><td>Equality</td></tr>
<tr><td>></td><td>Greater than operator</td></tr>
<tr><td>>=</td><td>Greater than or equal to operator</td></tr>
<tr><td>++</td><td>Increment</td></tr>
<tr><td>*</td><td>Indirection operator</td></tr>
<tr><td>!=</td><td>Inequality</td></tr>
<tr><td><<=</td><td>Left shift assignment operator, x<<=y, is the same as x=x<<y</td></tr>
<tr><td><</td><td>Less than operator</td></tr>
<tr><td><<</td><td>Left Shift operator</td></tr>
<tr><td><=</td><td>Less than or equal to operator</td></tr>
<tr><td>&&</td><td>Logical AND operator</td></tr>
<tr><td>!</td><td>Logical negation operator</td></tr>
<tr><td>||</td><td>Logical OR operator</td></tr>
<tr><td>%=</td><td>Modules assignment operator x%=y, is the same as x=x%y</td></tr>
<tr><td>%</td><td>Modules operator</td></tr>
<tr><td>*=</td><td>Multiplication assignment operator, x*=y, is the same as x=x*y</td></tr>
<tr><td>*</td><td>Multiplication operator</td></tr>
<tr><td>~</td><td>One's complement operator</td></tr>
<tr><td>>>=</td><td>Right shift assignment, x>>=y, is the same as x=x>>y</td></tr>
<tr><td>>></td><td>Right shift operator</td></tr>
<tr><td>-></td><td>Structure Pointer operation</td></tr>
<tr><td>-=</td><td>Subtraction assignment operator</td></tr>
<tr><td>-</td><td>Subtraction operator</td></tr>
<tr><td>sizeof</td><td>Determines size in bytes of operand</td></tr>
</table>

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					
!value ? expr: expr					
value = expr	value+=expr	value-=expr			
value*=expr	value/=expr	value%=expr			
value>>=expr	value<<=expr	value&=expr			
value^=expr	value|=expr	expr, expr			

BUILT-IN FUNCTIONS

Built-In Function List By Category			
RS232 I/O		**Parallel Slave I/O**	
getc()	p.78	setup_psp()	p.124
putc()	p.106	psp_input_full()	p.105
fgetc()	p.78	psp_output_full()	p.105
gets()	p.79	psp_overflow()	p.105
puts()	p.107	**I2C I/O**	
fgets()	p.79	i2c_start()	p.82
fputc()	p.106	i2c_stop()	p.83
fputs()	p.107	i2C_read	p.81
printf()	p.103	i2c_write()	p.81
kbhit()	p.88	i2c_poll()	p.81
fprintf()	p.103	**Processor Controls**	
set_uart_speed()	p.118	sleep()	p.134
perror()	p.102	reset_cpu()	p.111
assert()	p.64	restart_cause()	p.111
SPI two wire I/O		disable_interrupts()	p.72
setup_spi()	p.125	enable_interrupts()	p.73
spi_read()	p.135	ext_int_edge()	p.74
spi_write()	p.136	read_bank()	p.108
spi_data_is_in()	p.134	write_bank()	p.143
Discrete I/O		label_address()	p.89
output_low()	p.100	goto_address()	p.80
output_high()	p.100	**Bit/Byte Manipulation**	
output_float()	p.99	shift_right()	p.131
output_bit()	p.98	shift_left()	p.130
input()	p.84	rotate_right()	p.114
output_X()	p.101	rotate_left()	p.113
input_X()	p.85	bit_clear()	p.67
port_b_pullups()	p.102	bit_set()	p.67
set_tris_X()	p.117	bit_test()	p.68
Capture/Compare/PWM		swap()	p.141
setup_ccpX()	p.120	make8()	p.93
set_pwmX_duty()	p.115	make16()	p.94
Delays		make32()	p.95
delay_us()	p.71	**Analog Compare**	
delay_ms()	p.70	setup_comparator()	p.121
delay_cycles()	p.70		

Built-In Function List By Category… Continued			
Standard C Math		**Standard C Char**	
abs()	p.64	atoi()	p.66
acos()	p.64	atoi32()	p.66
asin()	p.64	atol()	p.66
atan()	p.65	atof()	p.65
ceil()	p.69	tolower()	p.142
cos()	p.69	toupper()	p.142
exp()	p.73	isalnum()	p.86
floor()	p.75	isalpha()	p.86
labs()	p.89	isamoung()	p.86
sinh()	p.134	isdigit()	p.86
log()	p.92	islower()	p.86
log10()	p.93	isspace()	p.86
pow()	p.103	isupper()	p.86
sin()	p.132	isxdigit()	p.86
cosh()	p.69	strlen()	p.138
tanh()	p.142	strcpy()	p.138
fabs()	p.75	strncpy()	p.138
fmod()	p.76	strcmp()	p.138
atan2()	p.65	stricmp()	p.138
frexp()	p.77	strncmp()	p.138
ldexp()	p.91	strcat()	p.138
modf()	p.97	strstr()	p.138
sqrt()	p.137	strchr()	p.138
tan()	p.142	strrchr()	p.138
Voltage Ref		strtok()	p.139
setup_vref()	p.128	strspn()	p.138
A/D Conversion		strcspn()	p.138
setup_adc_ports()	p.119	strpbrk()	p.138
setup_adc()	p.119	strlwr()	p.138
set_adc_channel()	p.114	sprintf()	p.136
read_adc()	p.107	**Standard C memory**	
		memset()	p.96
		memcpy()	p.95
		offsetof()	p.97
		offsetofbit()	p.97

Built-In Function List By Category… Continued			
Timers		**Internal EEPROM**	
setup_timer_X()	p.125	read_eeprom()	p.110
set_timer_X()	p.116	write_eeprom()	p.143
get_timer_X()	p.77	read_program_eeprom()	p.110
setup_counters()	p.122	write_program_eeprom()	p.144
setup_wdt()	p.129	read_calibration()	p.109
restart_wdt()	p.112		

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 the RS232:
// Ascertain failed in file : myfile.c at line : 56
```

Example files: None

Also see: #use rs232

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 too 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()

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: None

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()

CEIL( )

Syntax: result = ceil (***value***)

Parameters: ***value*** is a float

Returns: A float

Function: Computes the smallest integral value greater than the argument. Float(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()

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 0-255 or a constant 0-65535

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 0-255 or a constant 0-65535

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.

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: ex_sisr.c, ex_stwt.c

Also See: enable_interrupts(), #int_xxxx

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.

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: ex_sisr.c, ex_stwt.c

Also See: disable_enterrupts(), #int_xxxx

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 fvalue 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 error variable. The user can check the error 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 i 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()

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()

FLOOR( )

Syntax: result = floor (***value***)

Parameters: ***value*** is a float

Returns: 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

Functions: returns the floating point remainder of val1/val2 returns the value val1 - i*val2 for some integer 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

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()

Parameters: None

Returns: Timers 1 and 3 return a 16 bit int.
Timer 2 returns a 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

Requires: Nothing

Examples:

```
set_timer0(0);
while ( get_timer0() < 200 ) ;
```

Example Files: ex_stwt.c

Also See: set_timerx(), setup_timerx()

GETC()
GETCH()
GETCHAR()
FGETC()

Syntax: value = getc()
value = fgetc(stream)

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

Returns: A 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,strea
m=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

GETS( )
FGETS()

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

Parameters: ***string*** is a pointer to a 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_POLL( )

Syntax: i2c_poll()

Parameters: None

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_poll

I2C_READ()

Syntax: data = i2c_read();
or
data = i2c_read(***ack***);

Parameters: ***ack*** -Optional, defaults to 1.
0 indicates do not ack.
1 indicates to ack.

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.

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

See Also: i2c_start, i2c_write, i2c_stop, i2c_poll

I2C_START()

Syntax: i2c_start()

Parameters: None

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.

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_stop, i2c_write, i2c_read, i2c_poll, #use i2c

I2C_STOP()

Syntax: i2c_stop()

Parameters: None

Returns: undefined

Function: Issues a stop condition when in the I2C is in 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_WRITE( )

Syntax: i2c_write (***data***)

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

Returns: This function returns the ACK Bit.
0 means ACK, 1 means NO ACK.

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

---

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

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");
```

Example Files: EX_PULSE.C

Also See: input_x(), output_low(), output_high(), #use xxxx_io

INPUT_x()

Syntax:
value = input_a()
value = input_b()
value = input_c()
value = input_d()
value = input_e()

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: ex_psp.c

Also See: input(), output_x(), #USE xxxx_IO

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;
...
if( isamoung( x,
       "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ") )
   printf("The character is valid");
```

Example Files: ctype.h

Also See: isalnum(), isalpha(), isdigit(), isspace(), islower(), isupper(), isxdigit()

ISALNUM(char)
ISALPHA(char)
ISDIGIT(char)
ISLOWER(char)
ISSPACE(char)
ISUPPER(char)
ISXDIGIT(char)
ISCNTRL(x)
ISGRAPH(x)
ISPRINT(x)
ISPUNCT(x)

Syntax: value = isalnum(***datac***)
value = isalpha(***datac***)
value = isdigit(***datac***)
value = islower(***datac***)
value = isspace(***datac***)
value = isupper(***datac***)
value = isxdigit(***datac***)
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()

KBHIT()

Syntax: value = kbhit()

Parameters: None

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 is 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

LABEL_ADDRESS()

Syntax: value = label_address(label);

Parameters: label is a C label anywhere in the function

Returns: An 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.h

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()

LCD_SYMBOL( )

Syntax: lcd_symbol (***symbol***, ***b7_addr***, ***b6_addr***, ***b5_addr***, ***b4_addr***, ***b3_addr***, ***b2_addr***, ***b1_addr***, ***b0_addr***);

Parameters: ***symbol*** is a 8 bit constant.
bX_addr is a bit address representing the segment location to be used for bit X of symbol.

Returns: undefined

Function: Loads 8 bits into the segment data area for the LCD with each bit address specified. If bit 7 in symbol is set the segment at B7_addr is set, otherwise it is cleared. The same is true of all other bits in symbol. The B7_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()

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;
signed int exp;
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:
If "errno.h" is included then the domain and range errors are stored in the error variable. The user can check the error 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: Al devices

Requires: #include <math.h>

Examples:

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

Example Files: None

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

MAKE8()

Syntax: i8 = MAKE8(***var***, ***offset***)

Parameters: ***var*** is a 16 or 32 bit integer.
offset is a byte offset of 0,1,2 or 3.

Returns: An 8 bit integer

Function: Extracts the byte at offset from var. Same as: i8 = (((var >> (offset*8)) & 0xff) except it is done with a single byte move.

Availability: All devices

Requires: Nothing

Examples:

```
int32 x;
int y;

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

Example Files: None

Also see: make16(), make32()

MAKE16()

Syntax: i16 = MAKE16(***varhigh***, ***varlow***)

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

Returns: A 16 bit integer

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

Availability: All devices

Requires: Nothing

Examples:

```
long x;
int hi,lo;

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

Example Files: ltc1298.c

Also see: make8(), make32()

MAKE32()

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

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

Returns: A 32 bit integer

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

Availability: All devices

Requires: Nothing

Examples:

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

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

y=0x12;
z=0x4321;

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

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

Example Files: ex_freqc.c

Also see: make8(), make16()

MEMCPY( )

Syntax: memcpy (***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).

Availability: All devices.

Requires: Nothing

Examples:

```
memcpy(&structA, &structB, sizeof (structA));
memcpy(arrayA,arrayB,sizeof (arrayA));
memcpy(&structA, &databyte, 1);
```

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 n bytes of memory at destination with the value. Be aware that array names are pointers where other variable names and structure names are not (and therefore need a & before them).

Availability: All devices

Requires: Nothing

Examples:

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

Example Files: None

Also See: memcpy()

MODF()

Syntax: result= modf (***value***, & ***integral***)

Parameters: ***value*** and ***integral*** are floats

Returns: result is a float

Function: The mod 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

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;
           short daylight_savings;
    }

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

Example files: None

Also see: None

OUTPUT_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. ***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
```

Example Files: ex_extee.c with 9356.c

Also See: input(), output_low(), output_high(), output_float(), output_x(), #use xxxx_io

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

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(), #use xxxx_io

OUTPUT_HIGH( )

Syntax: output_high (***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: 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);
```

Example Files: ex_sqw.c

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

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);
```

Example Files: ex_sqw.c

Also See: input(), output_high(), output_float(), output_bit(), output_x(), #use xxxx_io

---

OUTPUT_A( )
OUTPUT_B( )
OUTPUT_C( )
OUTPUT_D( )
OUTPUT_E( )

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

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: None

PORT_B_PULLUPS( )

Syntax: port_b_pull-ups (***value***)

Parameters: ***value*** is TRUE or FALSE

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 kbd.c

Also See: input(), input_x(), output_float()

POW( )

Syntax: f = pow (***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: Nothing

PRINTF( )
FPRINTF()

Syntax: printf (***string***)
or
printf (***cstring***, ***values***...)
or

printf (***fname***, ***cstring***, ***values***...)
value = 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 %wt where w 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. t is the type and may be one of the following:

- C Character
- S String or character
- U Unsigned int
- x Hex int (lower case output)
- X Hex int (upper case output)
- D Signed int
- e Float in exp format
- f Float
- Lx Hex long int (lower case)
- LX Hex long int (upper case)
- lu unsigned decimal long
- ld signed decimal long
- % Just a %

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

* Result is undefined - Assume garbage.

Availability: All devices

Requires: #use rs232 (unless fname is used)

Examples:

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

Example Files: ex_admm.c, ex_lcdkb.c

Also See: atoi(), puts(), putc() , getc() (for a stream example)

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()

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 gets() defaults to STDOUT (the last USE RS232).

Availability: All devices

Requires: #use rs232

Examples:

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

```
putc(13);
```

Example Files: ex_tgetc.c

Also See: getc(), printf(), #USE RS232

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()

READ_ADC()

Syntax: value = read_adc()

Parameters: None

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:

#DEVCE	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: Nothing

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);
}
```

Example Files: ex_admm.c, ex_14kad.c

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

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 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:

```
// 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 14kcal.c

Also See:

READ_EEPROM( )

Syntax: value = read_eeprom (***address***)

Parameters: ***address*** is an 8 bit 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()

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 on PCM parts and 8 bits on PCH parts.

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()

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: Nothing

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: This function will return the reason for 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	PCH
Enable/Disable	#fuses	setup_wdt()

Timeout time	setup_wdt()	#fuses
restart	restart_wdt()	restart_wdt()

Availability: All devices

Requires: #fuses

Examples:

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

Example Files: ex_wdt.c

Also See: #fuses, setup_wdt()

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_right(), 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()

SET_PWM1_DUTY( )
SET_PWM2_DUTY( )

Syntax: set_pwm1_duty (***value***)
set_pwm2_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 = 520; // .000416/(16*(1/20000000))
set_pwm1_duty(duty);
```

Example Files: ex_pwm.c

Also See: setup_ccpX()

SET_RTCC( )
SET_TIMER0( )
SET_TIMER1( )
SET_TIMER2( )
SET_TIMER3( )

Syntax: set_timer0(value) or set_rtcc (value)
set_timer1(value)
set_timer2(value)
set_timer3(value)

Parameters: Timers 1 and 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

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()

SET_TRIS_A( )
SET_TRIS_B( )
SET_TRIS_C( )
SET_TRIS_D( )
SET_TRIS_E( )

Syntax: set_tris_a (***value***)
set_tris_b (***value***)
set_tris_c (***value***)
set_tris_d (***value***)
set_tris_e (***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 xxxx_IO

SET_UART_SPEED()

Syntax: set_uart_speed (***baud***)

Parameters: ***baud*** is a constant 100-115200 representing the number of bits per second.

Returns: undefined

Function: Changes the baud rate of the built-in hardware RS232 serial port at run-time.

Availability: This function is only available on devices with a built in UART.

Requires: #use rs232

Examples:

```
// Set baud rate based on setting
// of pins B0 and B1

switch( input_b() & 3 ) {
    case 0 : set_uart_speed(2400);   break;
    case 1 : set_uart_speed(4800);   break;
    case 2 : set_uart_speed(9600);   break;
    case 3 : set_uart_speed(19200);  break;
}
```

Example Files: loader.c

Also See: #USE RS232, putc(), getc()

SETUP_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

See Also: setup_adc_ports, set_adc_channel, read_adc, #device. The device .h file.

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()

SETUP_CCP1( )
SETUP_CCP2( )

Syntax: setup_ccp1 (***mode***)
setup_ccp2 (***mode***)

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

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 Files: ex_pwm.c, ex_ccpmp.c, ex_ccp1s.c

Also See: set_pwmX_duty()

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: None

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_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 ***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

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,
    ALL_LCD_PINS);
```

Example Files: ex_92lcd.c

Also See: lcd_symbol(), lcd_load()

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()

SETUP_SPI( )

Syntax: setup_spi (***mode***)

Parameters: ***modes*** 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()

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, 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_INTERVAL | T1_DIV_BY_8 );
```

Example Files: ex_patg.c

Also See: get_timer1(),

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 resets before an interrupt: (1 means one reset, 2 means 2, 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 include every 800ns,
```

```
// will overflow every 153.6us,
// and will interrupt every 460.3us.
```

Example Files: ex_pwm.c

Also See: get_timer2(), set_timer2()

SETUP_TIMER_3( )

Syntax: setup_timer_3 (***mode***)

Parameters: ***Mode*** may be one of the following constants from each group or'ed (via |) together:
- T3_DISABLED, T3_INTERNAL, T3_EXTERNAL, T3_EXTERNAL_SYNC, T3_DIV_BY_1, T3_DIV_BY_2, T3_DIV_BY_4, T3_DIV_BY_8

Returns: undefined

Function: Initializes timer 3. 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_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: None

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

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:

PCB/PCM PCH

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

Availability: All devices

Requires: #fuses, Constants are defined in the devices .h file.

Examples:

```
#fuses WDT_18MS  // PIC18 example, See
                 // restart_wdt for a PIC18 example
main() {
   setup_wdt(WDT_ON);
   while (TRUE) {
     restart_wdt();
     perform_activity();
   }
}
```

Example Files: ex_wdt.c

Also See: #fuses, restart_wdt()

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 with 9356.c

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

SHIFT_RIGHT( )

Syntax: shift_right (***address***, ***bytes***, ***value***)

Parameters: ***address*** is a pointer to memory, ***bytes*** is a count of the number
of bytes to work with, ***value*** is a 0 to 1 to be shifted in.

Returns: 0 or 1 for the bit shifted out

Function: Shifts a bit into an array or structure. The address may be an array identifier or an address to a structure (such as &data). Bit 0 of the lowest byte in RAM is treated as the LSB.

Availability: All devices

Requires: Nothing

Examples:

```
// reads 16 bits from pin A1, each bit is read
// on a low to high on pin A2
struct {
    byte time;
    byte command : 4;
    byte source  : 4;} msg;

for(i=0; i<=16; ++i) {
    while(!input(PIN_A2));
    shift_right(&msg,3,input(PIN_A1));
    while (input(PIN_A2)) ;}

// This shifts 8 bits out PIN_A0, LSB first.
for(i=0;i<8;++i)
```

```
output_bit(PIN_A0,shift_right(&data,1,0));
```

Example Files: ex_extee.c with 9356.c

Also See: shift_left(), rotate_right(), rotate_left(), <<, >>

SIN ( )
COS()
TAN()
ASIN()
ACOS()
ATAN()
SINH()
COSH()
TANH()
ATAN2()

Syntax:
val = sin (***rad***)
val = cos (***rad***)
val = tan (***rad***)
rad = asin (***val***)
rad1 = acos (***val***)
rad = atan (***val***)
rad2=atan2(***val***, ***val***)
result=sinh(***value***)
result=cosh(***value***)
result=tanh(***value***)

Parameters: ***rad*** is 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()

SPI_DATA_IS_IN()

Syntax: result = spi_data_is_in()

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:

```
while( !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_READ()

Syntax: value = spi_read (***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 supplies the clock 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 the other device supplies the clock 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_WRITE( )

Syntax: SPI_WRITE (***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.

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()

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: stdlib.h

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 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

STANDARD STRING FUNCTIONS
STRCAT()
STRCHR()
STRRCHR()
STRCMP()
STRNCMP()
STRICMP()
STRNCPY()
STRCSPN()
STRSPN()
STRLEN()
STRLWR()
STRPBRK()
STRSTR()

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

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

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)

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()

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()

STRCPY( )

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

Parameters: ***dest*** is a pointer to a RAM array of characters.
src may be either a pointer to a RAM array of characters or it may be a constant string.

Returns: undefined

Function: Copies a constant or RAM string to a RAM string. Strings are terminated with a 0.

Availability: All devices.

Requires: Nothing

Examples:

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

strcpy(string2,string);
```

Example Files: ex_str.c

Also See: strxxxx()

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( )

See: sin()

TANH()

See: SIN()

TOLOWER( )
TOUPPER( )

Syntax: result = tolower (***cvalue***)
result = toupper (***cvalue***)

Parameters: ***cvalue*** is a character

Returns: A 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_EEPROM( )

Syntax: write_eeprom (***address***, ***value***)

Parameters: ***address*** is a 8 bit 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

Also See: read_eeprom(), write_program_eeprom(), read_program_eeprom(), EX_EXTEE.c with CE51X.c, CE61X.c or CE67X.c.

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 on PCM parts and 8 bits on PCH parts.

Returns: undefined

Function: Writes to the specified program EEPROM area.

Availability: Only devices that allow writes to program memory.

Requires: Nothing

Examples:

```
write_program_eeprom(0,0x2800);
    //disables program
```

Example Files: ex_load.c, loader.c

Also See: read_program_eeprom(), read_eeprom(), write_eeprom()

STANDARD C DEFINITIONS

Standard C Definitions

Various header files are supplied with the compiler to provide standard C definitions and macros. The following is a list of the header files and the definitions within. See the files themselves or the C standard for more information.

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 an mutibyte 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

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.

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)

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

COMPILER ERROR MESSAGES

#ENDIF with no corresponding #IF
A numeric expression must appear here. The indicated item must evaluate to a number.

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.

Array dimensions must be specified
The [] notation is not permitted in the compiler. Specific dimensions must be used. For example A[5].

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 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.

Can't 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.

Constant too large, must be < 65536
As it says the constant is too big.

Define expansion is too large
A fully expanded DEFINE must be less than 255 characters. Check to be sure the DEFINE is not recursively defined.

Define syntax error
This is usually caused by a missing or mis-placed (or) within a define.

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 #define

The identifier in the #define has already been used in a previous #define. The 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

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
Check that the {and} match up correctly.

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.

Error in define syntax

Error text not in file
The error is a new error not in the error file on your disk. Check to be sure that the errors.txt file you are using came on the same disk as the version of software you are executing. Call CCS with the error number if this does not solve the problem.

Expect ;
Expect comma
Expect WHILE
Expect }
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 a]
Expecting a {
Expecting an =
Expecting an array
Expecting an expression
Expecting an identifier
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.

Expecting procedure name
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.

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 in #INCLUDE can not be opened
Check the filename and the current path. The file could not be opened.

Filename must start with " or <

Filename must terminate with " or >

Floating-point numbers not supported
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.

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);

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.

Invalid parameters to shift 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 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 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

Library in USE not found
The identifier after the USE is not one of the pre-defined libraries for the compiler. Check the spelling.

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);

Missing #ENDIF
A #IF was found without a corresponding #ENDIF.

Missing or invalid .REG 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).

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 MAIN() function found
All programs are required to have one function with the name main().

Not enough RAM for all variables
The program requires more RAM than is available. The memory map (ALT-M) will show variables allocated. The ALT-T will show 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.

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 '56 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 via ALT-T. 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 calling tree via ALT-T.

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.

Pointers to functions are not valid
Addresses cannot be created to functions.

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.

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 expression cannot evaluate to a number
A numeric result is required here and the expression used will not evaluate to a number.

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 #DEFINE statements
The internal compiler limit for the permitted number of defines has been reached. Call CCS to find out if this can be increased.

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 identifiers have been defined
The internal compiler limit for the permitted number of variables has been reached. Call CCS to find out if this can be increased.

Too many identifiers in program
The internal compiler limit for the permitted number of identifiers has been reached. Call CCS to find out if this can be increased.

Too many nested #INCLUDEs
No more than 10 include files may be open at a time.

Too many parameters
More parameters have been given to a function than the function was defined with.

Too many subscripts
More subscripts have been given to an array than the array was defined with.

Type is not defined
The specified type is used but not defined in the program. Check the spelling.

Type specification not valid for a function
This function has a type specifier that is not meaningful to a function.

Undefined identifier
The specified identifier is being used but has never been defined. Check the spelling.

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 type
The specified type is used but not defined in the program. Check the spelling.

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.

COMMON QUESTIONS AND ANSWERS

Questions	
How does one map a variable to an I/O port?	p.161
How do I directly read/write to internal registers?	p.181
Why does a program work with standard I/O but not with fast I/O?	p.163
Why does the generated code that uses BIT variables look so ugly?	p.164
Why is the RS-232 not working right?	p.165
How can I use two or more RS-232 ports on one PIC?	p.167
How does the PIC connect to a PC?	p.169
How do I get getc() to timeout after a specified time?	p.188
How do I do a printf to a string?	p.190
Why do I get an OUT OF ROM error when there seems to be ROM left?	p.170
What can be done about an OUT OF RAM error?	p.171
Why does the .LST file look out of order?	p.172
How is the TIMER0 interrupt used to perform an event at some rate?	p.173
How does the compiler handle converting between bytes and words?	p.174
How does the compiler determine TRUE and FALSE on expressions?	p.175
What are the restrictions on function calls from an interrupt function?	p.176
Why does the compiler use the obsolete TRIS?	p.178
How does the PIC connect to an I2C device?	p.179
Instead of 800, the compiler calls 0. Why?	p.180
Instead of A0, the compiler is using register 20. Why?	p.180
How can a constant data table be placed in ROM?	p.182
What is the format of floating point numbers?	p.184
Why does the compiler show less RAM than there really is?	p.185
What is an easy way for two or more PICs to communicate?	p.186
How do I write variables to EEPROM that are not a byte?	p.187
How do I put a NOP at location 0 for the ICD?	p.190
How do I make a pointer to a function?	p.191
How can I pass a variable to functions like OUTPUT_HIGH()?	p.189

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

Two methods are as follows:

```
#byte   PORTB  =  6
#define ALL_OUT 0
#define ALL_IN  0xff
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
}
```

Why does a program work with standard I/O but not with fast I/O?

First remember that the fast I/O mode does nothing except the I/O. The programmer must set the tri-state registers to establish the direction via SET_TRIS_X(). The SET_TRIS_X() function will set the direction for the entire port (8 bits). A bit set to 1 indicates input and 0 is an output. For example, to set all pins of port B to outputs except the B7 pin, use the following:

```
set_tris_b( 0x80 );
```

Secondly, be aware that fast I/O can be very fast. Consider the following code:

```
output_high( PIN_B0 );
output_low( PIN_B1 );
```

This will be implemented with two assembly instructions (BSF 6,0 and BCF 6,1). The microprocessor implements the BSF and BCF as a read of the entire port, a modify of the bit and a write back of the port. In this example, at the time that the BCF is executed, the B0 pin may not have yet stabilized. The previous state of pin B0 will be seen and written to the port with the B1 change. In effect, it will appear as if the high to B0 never happened. With standard and fixed I/O, this is not usually a problem since enough extra instructions are inserted to avoid a problem. The time it takes for a pin to stabilize depends on the load placed on the pin. The following is an example of a fix to the above problem:

```
output_high( PIN_B0 );
delay_cycles(1);   //Delay one instruction time
output_high( PIN_B1 );
```

The delay_cycles(1) will simply insert one NOP between the two I/O commands. At 20mhz a NOP is 0.2 us.

Why does the generated code that uses BIT variables look so ugly?

Bit variables (SHORT INT) are great for both saving RAM and for speed but only when used correctly. Consider the following:

```
int x,y;
short int bx, by;
x=5;
y=10;
bx=0;
by=1;
x = (x+by)-bx*by+(y-by);
```

When used with arithmetic operators (+ and - above), the BX and BY will be first converted to a byte internally: this is ugly. If this must be done, you can save space and time by first converting the bit to byte only once and saving the compiler from doing it again and again. For example:

```
z=by;
x = (x+z)-bx*z+(y-z);
```

Better, would be to avoid using bits in these kinds of expressions. Almost always, they can be rewritten more efficiently using IF statements to test the bit variables. You can make assignments to bits, use them in IFs and use the &&, || and ! operators very efficiently. The following will be implemented with great efficiency:

```
if (by || (bx && bz) || !bw)
z=0;
```

Remember to use ! not ~, && not & and || not | with bits. Note that the INPUT(...) function and some other built-in functions that return a bit follow the same rules.

For example do the following:

```
if ( !input( PIN_B0 ) )
```

NOT:

```
if( input( PIN_B0 ) == 0)
```

Both will work but the first one is implemented with one bit test instruction and the second one does a conversion to a byte and a comparison to zero.

Why is the RS-232 not working right?

1. The PIC® is Sending Garbage Characters.

A. Check the clock on the target for accuracy. Crystals are usually not a problem but RC oscillators can cause trouble with RS-232. Make sure the #USE DELAY matches the actual clock frequency.

B. Make sure the PC (or other host) has the correct baud and parity setting.

C. Check the level conversion. When using a driver/receiver chip, such as the MAX 232, do not use INVERT when making direct connections with resistors and/or diodes. You probably need the INVERT option in the #USE RS232.

D. Remember that PUTC(6) will send an ASCII 6 to the PC and this may not be a visible character. PUTC('A') will output a visible character A.

2. The PIC® is Receiving Garbage Characters.

A. Check all of the above.

3. Nothing is Being Sent.

A. Make sure that the tri-state registers are correct. The mode (standard, fast, fixed) used will be whatever the mode is when the #USE RS232 is encountered. Staying with the default STANDARD mode is safest.

B. Use the following main() for testing:

```
main() {
   while(TRUE)
       putc('U');
}
```

Check the XMIT pin for activity with a logic probe, scope or whatever you can. If you can look at it with a scope, check the bit time (it should be 1/BAUD). Check again after the level converter.

4. Nothing is being received.

First be sure the PIC® can send data. Use the following main() for testing:

```
main() {
```

```
    printf("start");
      while(TRUE)
         putc( getc()+1 );
}
```

When connected to a PC typing A should show B echoed back.
If nothing is seen coming back (except the initial "Start"), check the RCV pin on the PIC® with a logic probe. You should see a HIGH state and when a key is pressed at the PC, a pulse to low. Trace back to find out where it is lost.

5. The PIC® is always receiving data via RS-232 even when none is being sent.

A. Check that the INVERT option in the USE RS232 is right for your level converter. If the RCV pin is HIGH when no data is being sent, you should NOT use INVERT. If the pin is low when no data is being sent, you need to use INVERT.

B. Check that the pin is stable at HIGH or LOW in accordance with A above when no data is being sent.

C. When using PORT A with a device that supports the SETUP_PORT_A function make sure the port is set to digital inputs. This is not the default. The same is true for devices with a comparator on PORT A.

6. Compiler reports INVALID BAUD RATE.

A. When using a software RS232 (no built-in UART), the clock cannot be really slow when fast baud rates are used and cannot be really fast with slow baud rates. Experiment with the clock/baud rate values to find your limits.

B. When using the built-in UART, the requested baud rate must be within 3% of a rate that can be achieved for no error to occur. Some parts have internal bugs with BRGH set to 1 and the compiler will not use this unless you specify BRGH1OK in the #USE RS232 directive.

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);
   }
}
```

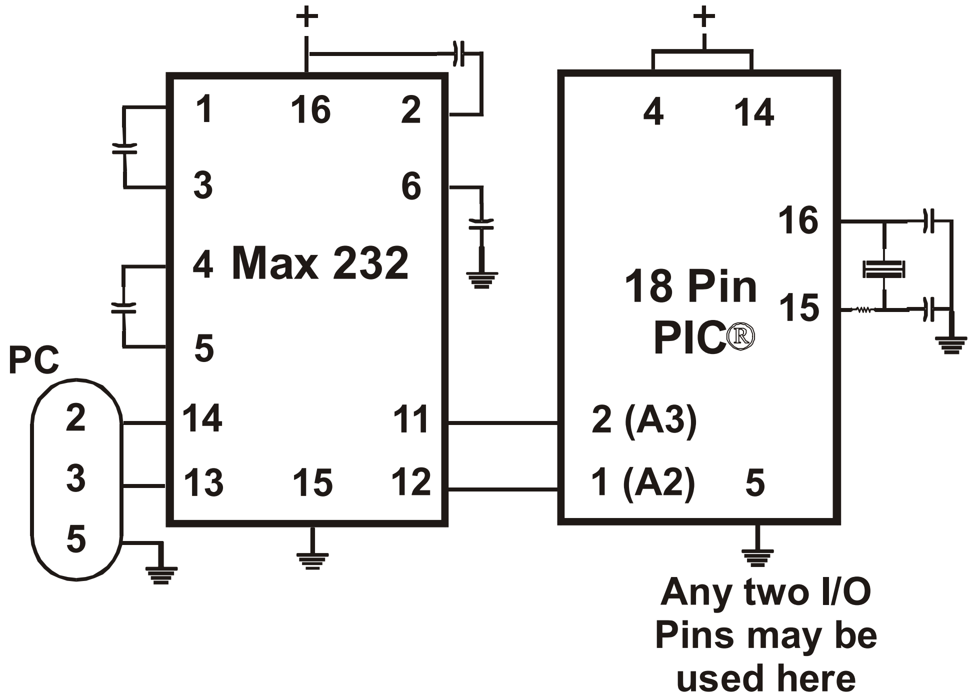
The following will do the same thing but is less readable:

```
main()  {
char c;
#USE RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1)
    printf("Online\n\r");
#USE RS232(BAUD=9600, #useXMIT=PIN_B2,RCV=PIN_B3)
    printf("Online\n\r");
    while(TRUE) {
#USE RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1)
    c=getc();
#USE RS232(BAUD=9600, XMIT=PIN_B2,RCV=PIN_B3)
    putc(c);
```

```
#USE RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1)
    putc(c);
}
```

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.



Why do I get an OUT OF ROM error when there seems to be ROM left?

The OUT OF ROM error can occur when a function will not fit into a segment. A function and all of its inline functions must fit into one hardware page. Sometimes decisions are made automatically by the linker. This will cause too many functions to be INLINE for a function to fit into a segment. To correct the problem, the user may need to use #SEPARATE to force a function to be separate. Consider the following example:

```
└─TEST.C
    MAIN ?614   RAM=5
      ├─DELAY_MS 0/19 RAM=1
      ├─READ_DATA (INLINE) RAM=5
      ├─PROCESS_DATA (INLINE) RAM=11
      ├─OUTPUT_DATA (INLINE) RAM=6
      └─PUTHEX (INLINE) RAM=2
          ├─PUTHEX1 0/18 RAM=2
          │   ├─ @PUTCHAR_9600_52_49  0/30 RAM=2
          │   └─ @PUTCHAR_9600_52_49  0/30 RAM=2
          └─PUTHEX1 0/18 RAM=2
              ├─ @PUTCHAR_9600_52_49  0/30 RAM=2
              └─ @PUTCHAR_9600_52_49  0/30 RAM=2
```

This example shows a main program with several INLINE functions that it calls. The resulting size of MAIN() is 614 locations and this will not fit into a 512 location page in the '56 device. The linker will put a ? in for the segment number since it would not fit in any segment. Note that the x/y notation is the page number (x) and number of locations (y). As a general rule, the linker will INLINE functions called only once to save stack space and in this program caused the function to get too large. The solution in this example will be to put a #SEPARATE before the declaration for PROCESS_DATA or maybe one of the other big functions called by MAIN(). The result might look like the following:

```
└─TEST.C
    MAIN ?406   RAM=5
      ├─DELAY_MS 0/19 RAM=1
      ├─READ_DATA (INLINE) RAM=5
      ├─PROCESS_DATA (INLINE) RAM=11
      ├─OUTPUT_DATA (INLINE) RAM=6
      └─PUTHEX (INLINE) RAM=2
          ├─PUTHEX1 0/18 RAM=2
          │   ├─ @PUTCHAR_9600_52_49  0/30 RAM=2
          │   └─ @PUTCHAR_9600_52_49  0/30 RAM=2
          └─PUTHEX1 0/18 RAM=2
              ├─ @PUTCHAR_9600_52_49  0/30 RAM=2
              └─ @PUTCHAR_9600_52_49  0/30 RAM=2
```

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. This is done automatically by the compiler whenever you use SHORT INT. The code size and ROM size will be smaller.

Finally, consider an external memory device to hold data not required frequently. An external 8 pin EEPROM or SRAM can be connected to the PIC® with just 2 wires and provide a great deal of additional storage capability. The compiler package includes example drivers for these devices. The primary drawback is a slower access time to read and write the data. The SRAM will have fast read and write with memory being lost when power fails. The EEPROM will have a very long write cycle, but can retain the data when power is lost.

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 a 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 a 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.

How is the TIMER0 interrupt used to perform an event at some rate?

The following is generic code used to issue a quick pulse at a fixed rate:

```
#include <16Cxx.H>
#use Delay(clock=15000000)
#define HIGH_START 114
byte seconds, high_count;
#INT_RTCC
clock_isr() {
  if(--high_count==0) {
     output_high(PIN_B0);
     delay_us(5);
     output_low(PIN_B0);
     high_count=HIGH_START;
  }
}
main() {
   high_count=HIGH_START;
   set_rtcc(0);
   setup_counters(RTCC_INTERNAL, RTCC_DIV_128);
   enable_interrupts(RTCC_ZERO);
   enable_interrupts(GLOBAL);
   while(TRUE);
}
```

In this program, the pulse will happen about once a second. The math is as follows:

The timer is incremented at (CLOCK/4)/RTCC_DIV.

In this example, the timer is incremented (15000000/4)/128 or 29297 times a second (34us).

The interrupt happens every 256 increments.

In this example, the interrupt happens 29297/256 or 114 times a second.

The interrupt function decrements a counter (HIGH_START times) until it is zero, then issues the pulse and resets the counter.

In this example, HIGH_START is 114 so the pulse happens once a second.
If HIGH_START were 57, the pulse would be about twice a second.

How does the compiler handle converting between bytes and words?

In an assignment such as:

```
bytevar = wordvar;
```

The most significant BYTE is lost. This is the same result as:

```
bytevar = wordvar & 0xff;
```

The following will yield just the most significant BYTE:

```
bytevar = wordvar >> 8;
```

Any arithmetic or relational expression involving both bytes and words will perform word operations, and treat the bytes as words with the top byte 0. For example:

```
wordvar= 0x1234;
bytevar= 0x34;
if(wordvar==bytevar)  //will be FALSE
```

Any arithmetic operations that only involve bytes will yield a byte result even when assigned to word.

For example:

```
bytevar1 = 0x80;
bytevar2 = 0x04;
wordvar = bytevar1 * bytevar2;
//wordvar will be 0
```

However, typecasting may be used to force word arithmetic:

```
wordvar = (long) bytevar1 * (long) bytevar2;
//wordvar will be 0x200
```

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 relation 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
```

What are the restrictions on function calls from an interrupt function?

Whenever interrupts are used, the programmer MUST ensure there will be enough stack space. Ensure the size of the stack required by the interrupt plus the size of the stack already used by main() wherever interrupts are enabled is less than 9. This can be seen at the top of the list file.

The compiler does not permit recursive calls to functions because the RISC instruction set does not provide an efficient means to implement a traditional C stack. All RAM locations required for a given function are allocated to a specific address at link time in such a way that RAM is re-used between functions not active at the same time. This prohibits recursion. For example, the main() function may call a function A() and A() may call B() but B() may NOT call main(), A() or B().

An interrupt may come in at any time, which poses a special problem. Consider the interrupt function called ISR() that calls the function A() just like main() calls A(). If the function A() is executing because main() called it and then the ISR() activates, recursion will have happened.

In order to prevent the above problem, the compiler will "protect" the function call to A() from main() by disabling all interrupts before the call to A() and restoring the interrupt state after A() returns. In doing so, the compiler can allow complete sharing of functions between the main program and the interrupt functions.

The programmer must take the following special considerations into account:

1. In the above example, interrupts will be disabled for the entire execution of A(). This will increase the interrupt latency depending on the execution time of A().

2. If the function A() changes the interrupts using ENABLE/DISABLE _INTERRUPTS then the effect may be lost upon the return from A(), since the entire INTCON register is saved before A() is called and restored afterwards. Furthermore, if the global interrupt flag is enabled in A(), the program may execute incorrectly.

3. A program should not depend on the interrupts being disabled in the above situation. The compiler may NOT disable interrupts when the function or any function it calls requires no local RAM.

4. The interrupts may be disabled, as described above for internal compiler functions called by the same manor. For example, multiplication invoked by a

simple * may have this effect.

Why does the compiler use the obsolete TRIS?

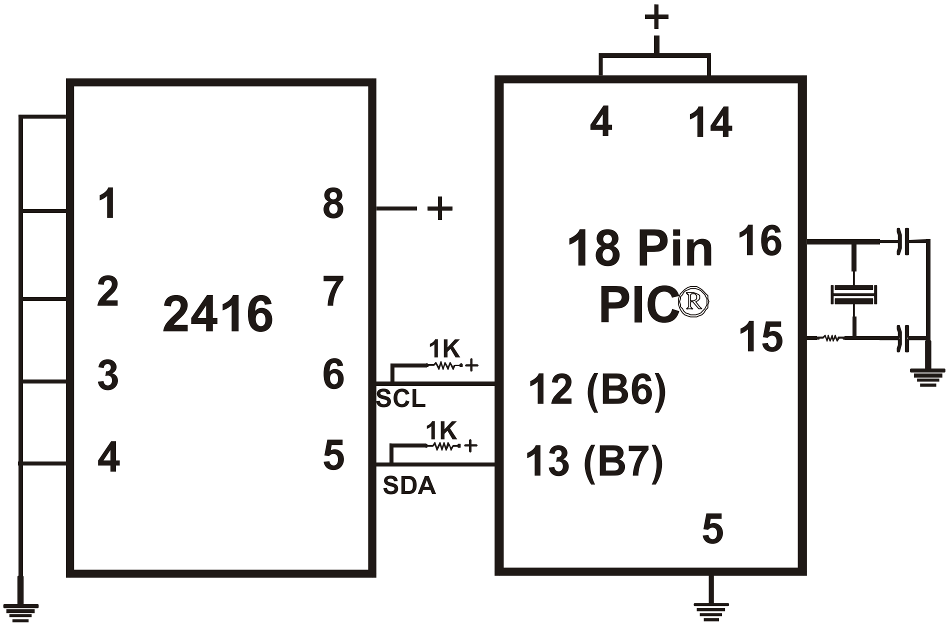
The use of TRIS causes concern for some users. The Microchip data sheets recommend not using TRIS instructions for upward compatibility. If you had existing ASM code and it used TRIS then it would be more difficult to port to a new Microchip part without TRIS. C does not have this problem, however; the 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 your part and the compiler will not use it.

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.



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 you will see:

```
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 you will see:

```
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.

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 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 the RB interrupt be used to detect a button press?

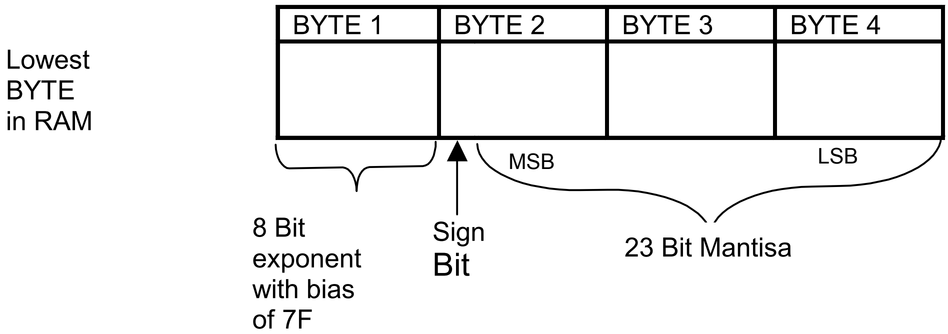
The RB interrupt will happen when there is any change (input or output) on pins B4-B7. There is only one interrupt and the PIC® does not tell you which pin changed. The programmer must determine the change based on the previously known value of the port. Furthermore, a single button press may cause several 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. Don't process further changes on the pin.

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 PCONVERT that will provide easy conversion to/from decimal, hex and float in a small window in Windows. 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

Lowest BYTE in RAM

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 you need 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. You can switch to larger pointers for full RAM access (this takes more ROM). In PCB add *=8 to the #device and in PCM/PCH add *=16 to the #device.

Example:

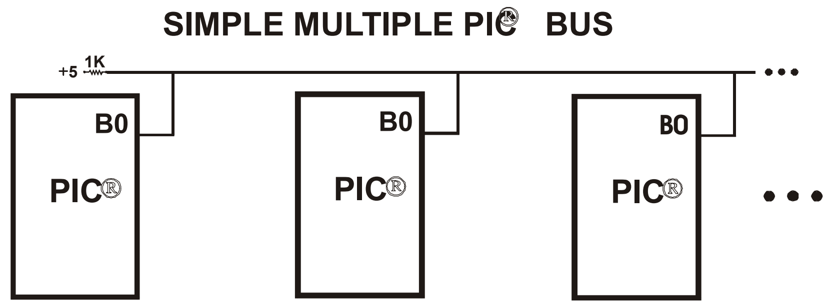
#DEVICE PIC16C77 *=16

or

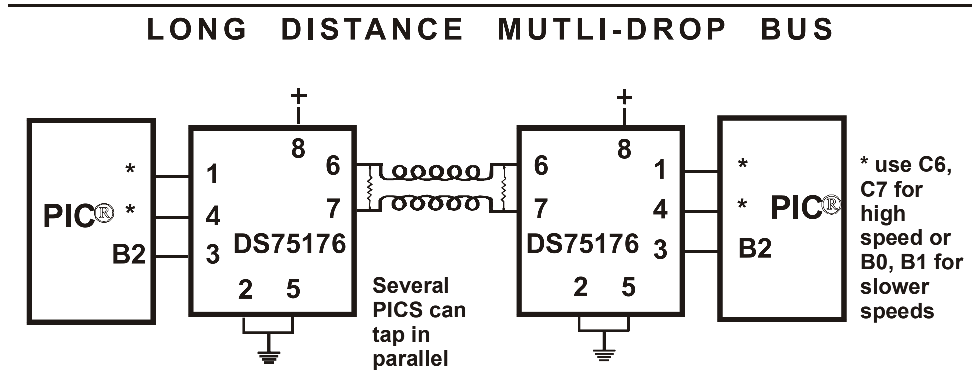
#include <16C77.h>
#device *=16

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.




```
#USE RS232 (baud=9600, float_high, bits=9, xmit=PIN_B0, rcv=PIN_B0)
```




```
#USE RS232 (baud=9600, bits=9, xmit=PIN_*, RCV=PIN_*, enable=PIN_B2)
```

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 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 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 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 #DEIVCE.

For example:

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

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.

```
char string[20];
byte stringptr=0;

tostring(char c) {
   if(c=='\f')
     stringptr=0;
   else
     string[stringptr++]=c;
   string[stringptr]=0;
}

main() {
   float f;

   f=12.345;

   printf(tostring,"\f%6.3f",f);

}
```

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 much time does 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.

8 bit add	<1 us
8 bit multiply	9 us
8 bit divide	20 us
16 bit add	2 us
16 bit multiply	48 us
16 bit divide	65 us
32 bit add	5 us
32 bit multiply	138 us
32 bit divide	162 us
float add	32 us
float multiply	147 us
float divide	274 us
exp()	1653 us
ln()	2676 us
sin()	3535 us

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--)
```

EXAMPLE PROGRAMS

A large number of example programs are included on the disk. 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 SIO.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
An analog to digital program with calibration for the PIC14000

EX_1920
Uses a Dallas DS1920 button to read temperature

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

EX_92LCD
Uses a PIC16C92x chip to directly drive LCD glass

EX_AD12
Shows how to use an external 12 bit A/D converter

EX_ADMM
A/D Conversion example showing min and max analog readings

EX_CCP1S
Generates a precision pulse using the PIC® CCP module

EX_CCPMP

Uses the PIC® CCP module to measure a pulse width

EX_COMP
Uses the analog comparator and voltage reference available on some PIC®s

EX_CRC
Calculates CRC on a message showing the fast and powerful bit operations

EX_CUST
Change the nature of the compiler using special preprocessor directives

EX_FIXED
Shows fixed point numbers

EX_DPOT
Controls an external digital POT

EX_DTMF
Generates DTMF tones

EX_ENCOD
Interfaces to an optical encoder to determine direction and speed

EX_EXPIO
Uses simple logic chips to add I/O ports to the PIC®

EX_EXTEE
Reads and writes to an external EEPROM

EX_FLOAT
Shows how to use basic floating point

EX_FREQC
A 50 mhz frequency counter

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

EX_INTEE
Reads and writes to the PIC® internal EEPROM

EX_LCDKB
Displays data to an LCD module and reads data fro keypad

EX_LCDTH
Shows current, min and max temperature on an LCD

EX_LED
Drives a two digit 7 segment LED

EX_LOAD
Serial boot loader program for chips like the 16F877

EX_MACRO
Shows how powerful advanced macros can be in C

EX_PATG
Generates 8 square waves of different frequencies

EX_PBUSM
Generic PIC® to PIC® message transfer program over one wire

EX_PBUSR
Implements a PIC® to PIC® shared RAM over one wire

EX_PBUTT
Shows how to use the B port change interrupt to detect pushbuttons

EX_PGEN
Generates pulses with period and duty switch selectable

EX_PLL
Interfaces to an external frequency synthesizer to tune a radio

EX_PSP
Uses the PIC® PSP to implement a printer parallel to serial converter

EX_PULSE
Measures a pulse width using timer0

EX_PWM
Uses the PIC® CCP module to generate a pulse stream

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

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

EX_RTC
Sets and reads an external Real Time Clock using RS232

EX_RTCLK
Sets and reads an external Real Time Clock using an LCD and keypad

EX_SINE
Generates a sine wave using a D/A converter

EX_SISR
Shows how to do RS232 serial interrupts

EX_SLAVE
Simulates an I2C serial EEPROM showing the PIC® slave mode

EX_SPEED
Calculates the speed of an external object like a model car

EX_SPI
Communicates with a serial EEPROM using the H/W SPI module

EX_SQW
Simple Square wave generator

EX_SRAM
Reads and writes to an external serial RAM

EX_STEP
Drives a stepper motor via RS232 commands and an analog input

EX_STR
Shows how to use basic C string handling functions

EX_STWT
A stop Watch program that shows how to user a timer interrupt

EX_TANK
Uses trig functions to calculate the liquid in an odd shaped tank

EX_TEMP

Displays (via RS232) the temperature from a digital sensor

EX_TGETC
Demonstrates how to timeout of waiting for RS232 data

EX_TONES
Shows how to generate tones by playing "Happy Birthday"

EX_TOUCH
Reads the serial number from a Dallas touch device

EX_USB
Implements a USB device on the PIC16C765

EX_VOICE
Self learning text to voice program

EX_WDT
Shows how to use the PIC® watch dog timer

EX_X10
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

4128.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

AD7715.C
A/D Converter functions

AD8400.C
Digital POT functions

AT25256.C
Serial EEPROM functions

CE51X.C
Functions to access the 12CE51x EEPROM

CE62X.C
Functions to access the 12CE62x EEPROM

CE67X.C
Functions to access the 12CE67x EEPROM

CTYPE.H
Definitions for various character handling functions

DS1302.C
Real time clock functions

DS1621.C
Temperature functions

DS1868.C
Digital POT functions

FLOATEE.C
Functions to read/write floats to an EEPROM

INPUT.C
Functions to read strings and numbers via RS232

KBD.C
Functions to read a keypad

LCD.C
LCD module functions

LOADER.C
A simple RS232 program loader

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

STDIO.H
Not much here - Provided for standard C compatibility

STDLIB.H
String to number functions

STRING.H
Various standard string functions

TONES.C
Functions to generate tones

TOUCH.C
Functions to read/write to Dallas touch devices

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);
}
```

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.

P.O. Box 2452
Brookfield, WI 53008

Index

#

#ASM 25
#BIT 27, 28
#BYTE 28
#CASE 28, 29
#DEFINE 29, 30
#DEVICE 30, 31
#ELIF 33, 34
#ELSE 33, 34, 35
#ENDASM 25, 26
#ENDIF 34, 35
#ERROR 31, 32
#FUSES 33
#ID Checksum 33
#ID Filename 33
#ID number 33
#IF expr 33
#IFDEF 34, 35
#IFNDEF 34, 35
#INCLUDE 35
#INLINE 36
#INT_AD 37
#INT_ADOF 36
#INT_BUSCOL 36
#INT_BUTTON 36
#INT_CCP1 36
#INT_CCP2 36
#INT_COMP 36
#INT_DEFAULT 37, 38
#INT_EEPROM 36
#INT_EXT 36
#INT_EXT1 36
#INT_EXT2 36
#INT_GLOBAL 38
#INT_I2C 36
#INT_LCD 36
#INT_LOWVOLT 36
#INT_PSP 36

#INT_RB.......... 36
#INT_RC 36
#INT_RDA.......... 36
#INT_RTCC 37
#INT_SSP 36
#INT_TBE.......... 36
#INT_TIMER0 36
#INT_TIMER1 36
#INT_TIMER2 36
#INT_TIMER3 36
#INT_xxxx 37
#LIST.......... 39
#LOCATE.......... 39
#NOLIST 39, 40
#OPT.......... 40
#ORG 40, 41
#PRAGMA.......... 43
#PRIORITY 43
#RESERVE 43, 44
#ROM.......... 44
#SEPARATE.......... 44
#TYPE.......... 45
#UNDEF.......... 46
#USE I2C 47
#USE RS232.......... 48
#USE DELAY.......... 46
#USE FAST_IO.......... 46
#USE FIXED_IO 47
#USE STANDARD_IO 50
#ZERO_RAM 50

_

_ _ DATE_ _.......... 29
_ _ PCH _ _.......... 42
_ _ TIME _ _.......... 45
_ _DEVICE_ _.......... 31
_ _PCB_ _.......... 41
_ _PCM_ _ 42
__FILE__.......... 32
__LINE__ 39

A

ABS 63

ACOS 64, 123, 124
ASIN 64, 123, 124
ASSERT 64
ATAN 65, 123, 124
ATAN2 .... 65, 124
ATOF 65
ATOI ATOL 65

B

BIT_CLEAR 66
BIT_SET 67
BIT_TEST 67
Break 57
Built-In Function List By Category 60
Built-In Functions 60

C

C Statements and Expressions 54
CEIL 68
char_bit 135
Code Examples 179
Comment 54
Common Questions and Answers 146
Compile Options 17
Compiler Error Messages 137
Continue 57
COS 68, 123
COSH 68, 124

D

Data Definitions 51
dbl_dig 135
dbl_epsilon 135
dbl_mant_dig 135
dbl_max 135
dbl_max_10_exp 135
dbl_max_exp 135
dbl_min 135
dbl_min_10_exp 135
dbl_min_exp 135
DELAY_CYCLES 69
DELAY_MS 69
DELAY_US 70

Device Calibration Data 13
Direct Device Programming 12
Directories 12
DISABLE_INTERRUPTS 71
Do 56

E

ENABLE_INTERRUPTS 71
Example Programs 172
EXP 72
Expr 57
Expressions 58, 59
EXT_INT_EDGE 73

F

FABS 73
FGETC 76
FGETS 77
File Formats 12
File Menu 14
float.h 135
FLOOR 74
flt_dig 135
flt_epsilon 135
flt_mant_dig 135
flt_max 135
flt_max_10_exp 135
flt_max_exp 135
flt_min 135
flt_min_10_exp 135
flt_min_exp 135
flt_radix 135, 136
FMOD 74
For 56
FPRINTF 98
FPUTC 100
FPUTS 101
FREXP 75
Function Definition 52, 53

G

GET_RTCC 75
GET_TIMER0 75

GET_TIMER1 75
GET_TIMER2 75
GET_TIMER3 75
GET_TIMERx 75
GETC 76
GETCH 76
GETCHAR 76
GETS 77
Goto 57
GOTO_ADDRESS 78

H

Help Menu 21
How are type conversions handled? 170
How can a constant data table be placed in ROM? 162
How can I pass a variable to functions like OUTPUT_HIGH ? 168
How can I use two or more RS-232 ports on one PIC? 152
How can the RB interrupt be used to detect a button press? 163
How do I directly read/write to internal registers? 162
How do I do a printf to a string? 169
How do I get getc to timeout after a specified time? 167
How do I make a pointer to a function? 169
How do I put a NOP at location 0 for the ICD? 168
How do I write variables to EEPROM that are not a byte? 166
How does one map a variable to an I/O port? 147
How does the compiler determine TRUE and FALSE on expressions? 159
How does the compiler handle converting between bytes and words? 158
How does the PIC connect to a PC? 153
How does the PIC connect to an I2C device? 161
How is the TIMER0 interrupt used to perform an event at some rate? 157
How much time do math operations take? 170

I

I2C_POLL 78
I2C_READ 79
I2C_START 80
I2C_STOP 80
I2C_WRITE 81
IF Statement 55
INPUT 81
INPUT_A 82
INPUT_B 82
INPUT_C 82

INPUT_D ... 82
INPUT_E ... 82
INPUT_x ... 82
Installation ... 10
Instead of 800 the compiler calls 0. Why? ... 161
Instead of A0 the compiler is using register 20. Why? ... 161
int_max ... 135
int_min ... 135
Invoking the Command Line Compiler ... 10
ISALNUM char ... 83
ISALPHA ... 83
ISAMOUNG ... 82
ISCNTRL ... 83
ISDIGIT ... 83
ISGRAPH ... 83
ISLOWER ... 83
ISPRINT ... 83
ISPUNCT ... 83
ISSPACE ... 83
ISUPPER ... 83
ISXDIGIT ... 83

J

jmp_buf ... 136

K

KBHIT ... 84

L

Label ... 57
LABEL_ADDRESS ... 85
LABS ... 86
LCD_LOAD ... 86
LCD_SYMBOL ... 87
ldbl_dig ... 135
ldbl_epsilon ... 135
ldbl_mant_dig ... 135
ldbl_max ... 135
ldbl_max_10_exp ... 135
ldbl_max_exp ... 135
ldbl_min ... 135
ldbl_min_10_exp ... 135
ldbl_min_exp ... 135

LDEXP........87, 88
limits.h........135
LOG........88
LOG10........89
long_max........135
long_min........135
longjmp........136

M

MAKE16........90
MAKE32........90
MAKE8........89
mb_len_max........135
MEMCPY........91
MEMSET........92
MODF........92
MPLAB Integration........11

N

null........136

O

OFFSETOF........93
OFFSETOFBIT........93
Operators........59
Options Menu........16
OUTPUT_A........96
OUTPUT_B........96
OUTPUT_BIT........93
OUTPUT_C........96
OUTPUT_D........96
OUTPUT_E........96
OUTPUT_FLOAT........94
OUTPUT_HIGH........95
OUTPUT_LOW........95
Overview........9

P

PCB PCM and PCH Overview........9
  PCB........9
  PCH........9
  PCM........9

PCW Editor C Features.......... 15
PCW Editor Keys 21
PCW IDE.......... 14
PCW Overview.......... 9
PERROR.......... 96
PORT_B_PULLUPS 97
POW.......... 97
Pre-Processor 23
Pre-Processor Command Summary 23
Pre-Processor Directives 25
PRINTF 98
Program Syntax 54
Project Menu 14
Project Wizard.......... 23
PSP_INPUT_FULL 99
PSP_OUTPUT_FULL 99
PSP_OVERFLOW 99
ptrdiff_t 136
PUTC.......... 100
PUTCHAR.......... 100
PUTS.......... 101

R

READ_ADC.......... 101
READ_BANK 102
READ_CALIBRATION 103
READ_EEPROM.......... 104
READ_PROGRAM_EEPROM.......... 104
Reference Parameters 53
RESET_CPU.......... 105
RESTART_CAUSE 105
RESTART_WDT 106
Return.......... 56
ROTATE_LEFT.......... 107
ROTATE_RIGHT 107

S

schar_max.......... 135
schar_min.......... 135
SET_ADC_CHANNEL 108
SET_PWM1_DUTY.......... 108
SET_PWM2_DUTY.......... 108
SET_RTCC 109

SET_TIMER0 109
SET_TIMER1 109
SET_TIMER2 109
SET_TIMER3 109
SET_TRIS_A 110
SET_TRIS_B 110
SET_TRIS_C 110
SET_TRIS_D 110
SET_TRIS_E 110
SET_UART_SPEED 111
setjmp 136
setjmp.h 136
SETUP_ADC mode 111
SETUP_ADC_PORTS 112
SETUP_CCP1 113
SETUP_CCP2 113
SETUP_COMPARATOR 114
SETUP_COUNTERS 115
SETUP_LCD 115
SETUP_PSP 116
SETUP_SPI 117
SETUP_TIMER_0 .... 117, 118
SETUP_TIMER_1 118
SETUP_TIMER_2 119
SETUP_TIMER_3 119
SETUP_VREF 120
SETUP_WDT 120
SHIFT_LEFT 121
SHIFT_RIGHT 122
shrt_max 135
shrt_min 135
SIN 123
SINH 124
size_t 136
SLEEP 125
Software License Agreement 183
SPI_DATA_IS_IN 125
SPI_READ 125, 126
SPI_WRITE 126
SPRINTF 127
SQRT 127
Standard C Definitions 134
STANDARD STRING FUNCTIONS 128

Statements 54, 55
stddef.h 136
Stmt 58
STRCAT 128
STRCHR 128
STRCMP 128
STRCPY 131
STRICMP 128
STRLEN 128
STRLWR 128
STRNCMP 128
STRNCPY 128
STRPBRK 128
STRRCHR 128
STRSPN 128
STRSTR 128
STRTOK 129
SWAP 131
Switch 56

T

TAN 123, 124, 132
TANH 124, 132
Technical Support 10
TOLOWER 132
Tools Menu 19
TOUPPER 132
Trigraph Sequences 54

U

uchar_max 135
uchar_min 135
uint_max 135
ulong_max 135
ushrt_max 135
Utility Programs 13

V

View Menu 18

W

wchar_t 136

What are the restrictions on function calls from an interrupt function?..... 159
What can be done about an OUT OF RAM error? 155
What is an easy way for two or more PICs to communicate? 165
What is the format of floating point numbers? 164
While 55
Why do I get an OUT OF ROM error when there seems to be ROM left? 154
Why does a program work with standard I/O but not with fast I/O?.......... 149
Why does the .LST file look out of order?... 156
Why does the compiler show less RAM than there really is? 164
Why does the compiler use the obsolete TRIS?.. 160
Why does the generated code that uses BIT variables look so ugly? 149
Why is the RS-232 not working right?.. 150
WRITE_BANK.. 132
WRITE_EEPROM 133
WRITE_PROGRAM_EEPROM 134