# CCS C Compiler Manual

## PCB, PCM, PCH, and PCD


**March 2019**

# Table of Contents

# OVERVIEW

## PCB, PCM, PCH and PCD

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

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

The compiler can efficiently implement normal C constructs, input/output operations, and bit twiddling operations. All normal C data types are supported along with pointers to constant arrays, fixed point decimal, and arrays of bits.
*[PCD]* Special built in functions to perform common functions in the MPU with ease.

*[PCD]* Extended constructs like bit arrays, multiple address space handling and effective implementation of constant data in Rom make code generation very effective.

IDE Compilers (PCW, PCWH and PCWHD) have the exclusive C Aware integrated development environment for compiling, analyzing and debugging in real-time. Other features and integrated tools can be viewed here.

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

## Installation

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

If the CD does not auto start run the setup program in the root directory.

For help answering the version questions see the "Directories" Help topic.

Key Questions that may come up:

**Keep Settings** - Unless you are having trouble select this
**Link Compiler Extensions** - If you select this the file extensions like .c will start the compiler IDE when you double click on files with that extension. .hex files start the CCSLOAD program. This selection can be change in the IDE.

**Install MP LAB Plug In** - If you plan to use MPLAB and you don't select this you will need to download and manually install the Plug-In.

**Install ICD2, ICD3**...drivers-select if you use these microchip ICD units.

**Delete Demo Files** - Always a good idea

**Install WIN8 APP**- Allows you to start the IDE from the Windows8 and Windows10 Start Menus.

## Technical Support

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

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

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

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

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

## Directories

The compiler will search the following directories for Include files.

- Directories listed on the command line
- Directories specified in the **.CCSPJT** file (edit in the IDE under **Options>Project>Include**)
- Directories specified in the **ccs.ini** file found using **Start>All Programs>PICC>User Data Dir**
- The same directory as the source.directories in the **ccsc.ini** file

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

The compiler itself is a DLL file. The DLL files are in a DLL directory by default in **\PICC\DLL\5.xxx**.

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

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

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

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

## File Formats

**.c** - This is the source file containing user C source code.

**.h** - These are standard or custom header files used to define pins, register, register bits, functions and preprocessor directives.

**.pjt** - This is the older pre- Version 5 project file which contains information related to the project.

**.ccspjt** - This is the project file which contains information related to the project.

**.lst** - This is the listing file which shows each C source line and the associated assembly code generated for that line.

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

**CCS Basic** - Standard assembly

**with Opcodes** - Includes the HEX opcode for each instruction

**Old Standard** -

**Symbolic** - Shows variable names instead of addresses

**Mach code** - Includes the HEX opcode for each instruction

**SRF names** - Instead of an address, a name is used. For example, instead of 044, will show CORCON

**Symbols** - Shows variable names instead of addresses

**Interpret** - Adds a pseudo code interpretation to the right of assembly instruction to help understand the operation. For example: LSR W4,#8,W5 : W5=W4>>8

**.sym** - This is the symbol map which shows each register location and what program variables are stored in each location.

**.sta** - The statistics file shows the RAM, ROM, and STACK usage. It provides information on the source codes structural and textual complexities using Halstead and McCabe metrics.

**.tre** - The tree file shows the call tree. It details each function and what functions it calls along with the ROM and RAM usage for each function.

**.hex** - The compiler generates standard HEX files that are compatible with all programmers. The compiler can output 8-bet hex, 16-bit hex, and binary files.

**.cof** - This is a binary containing machine code and debugging information. The debug files may be output as Microchip .COD file for MPLAB 1-5, Advanced Transdata .MAP file, expanded .COD file for CCS debugging or MPLAB 6 and up .xx .COF file. All file formats and extensions may be selected via Options File Associations option in Windows IDE.

**.cod** - This is the binary file containing debug information.

**.rtf** - The output of the Documentation Generator is exported in a Rich Text File format which can be viewed using the RTF editor or Wordpad.

**.rvf** - The Rich View Format is used by the RTF Editor within the IDE to view the Rich Text File.

**.dgr** - The .DGR file is the output of the flowchart maker.

**.esym or .xsym** - These files are generated for the IDE users. The file contains Identifiers and Comment information. This data can be used for automatic documentation generation and for the IDE helpers.

**.o** - Relocatable object file.

**.osym** - This file is generated when the compiler is set to export a relocatable object file. This file is a .sym file for just the one unit.

**.err** - Compiler error file.

**.ccsload** - Used to link Windows Apps to CCSLoad

**.ccssiow** - Used to link WindowsApps to Serial Port Monitor

## Invoking the Command Line Compiler

The command line compiler is invoked with the following command:

```
CCSC   [options]   [cfilename]
```

Valid options:

| **+FB** | **Select PCB (12 bit)** | **-D** | **Do not create debug file** |
|---|---|---|---|
| **+FM** | Select PCM (14 bit) | +DS | Standard .COD format debug file |
| **+FH** | Select PCH (PIC18XXX) | +DM | .MAP format debug file |
| **+Yx** | Optimization level x (0-9) | +DC | Expanded .COD format debug file |
| **+FD** | Select PCD (dsPIC30/dsPIC33/PIC24) | +DF | Enables the output of an COFF debug file. |
| **+FS** | Select SXC (SX) | +EO | Old error file format |
| **+ES** | Standard error file | -T | Do not generate a tree file |
| **+T** | Create call tree (.TRE) | -A | Do not create stats file (.STA) |
| **+A** | Create stats file (.STA) | -EW | Suppress warnings (use with +EA) |
| **+EW** | Show warning messages | -E | Only show first error |
| **+EA** | Show all error messages and all warnings | +EX | Error/warning message format uses GCC's "brief format" (compatible with GCC editor environments) |

The xxx in the following are optional. If included it sets the file extension:

| **+LNxxx** | **Normal list file** | **+O8xxx** | **8-bit Intel HEX output file** |
|---|---|---|---|
| **+LSxxx** | MPASM format list file | +OWxxx | 16-bit Intel HEX output file |
| **+LOxxx** | Old MPASM list file | +OBxxx | Binary output file |
| **+LYxxx** | Symbolic list file | -O | Do not create object file |
| **-L** | Do not create list file | | |
| | | | |
| **+P** | Keep compile status window up after compile | | |
| **+Pxx** | Keep status window up for xx seconds after compile | | |
| **+PN** | Keep status window up only if there are no errors | | |
| **+PE** | Keep status window up only if there are errors | | |
| | | | |

| | |
|---|---|
| **+Z** | Keep scratch files on disk after compile |
| **+DF** | COFF Debug file |
| **I+="..."** | Same as I="..." Except the path list is appended to the current list |
| **I="..."** | Set include directory search path, for example:<br>I="c:\picc\examples;c:\picc\myincludes"<br>If no I= appears on the command line the .PJT file will be used to supply the include file paths. |
| **out="dir"** | Use this directory for output files |
| **-P** | Close compile window after compile is complete |
| **+M** | Generate a symbol file (.SYM) |
| **-M** | Do not create symbol file |
| **+J** | Create a project file (.PJT) |
| **-J** | Do not create PJT file |
| **+ICD** | Compile for use with an ICD |
| **#xxx="yyy"** | Set a global #define for id xxx with a value of yyy, example:<br>#debug="true" |
| **+Gxxx="yyy"** | Same as #xxx="yyy" |
| **+?** | Brings up a help file |
| **-?** | Same as +? |
| | |
| **+STDOUT** | Outputs errors to STDOUT (for use with third party editors) |
| **+SETUP** | Install CCSC into MPLAB (no compile is done) |
| **sourceline=** | Allows a source line to be injected at the start of the source file.<br>Example: CCSC +FM myfile.c sourceline="#include <16F887.h>" |
| **+V** | Show compiler version (no compile is done) |
| **+Q** | Show all valid devices in database (no compile is done) |

A / character may be used in place of a + character. The default options are as follows:
+FM +ES +J +DC +Y9 -T -A +M +LNlst +O8hex -P -Z

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

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

Examples:

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

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

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

## Menu

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

## Editor Tabs

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

## Slide Out Windows

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

## Editor

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

## Debugging Windows

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

## Status Bar

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

## Output Messages

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

# PROGRAM SYNTAX

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

## Comment

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

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

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

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

```
    05/27.06   Fixed Flashing problem
*/
```

**Variable Comments**
A variable comment is a comment that appears immediately after a variable declaration. For example:

```
int seconds; // Number of seconds since last entry
long day,     // Current day of the month,   /* Current Month */
long year;      // Year
```

**Function Comments**
A function comment is a comment that appears just before a function declaration. For example:

```
// The following function initializes outputs
void function_foo()
{
        init_outputs();
}
```

**Function Named Comments**
The named comments can be used for functions in a similar manner to the Global Comments. These comments appear before the function, and the names are exported as-is to the documentation generator.
For example:

```
//*PURPOSE This function displays data in BCD format
void display_BCD( byte n)
{
        display_routine();
 }
```

## Trigraph Sequences

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

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

| | |
|---|---|
| ??! | \| |
| ??> | } |
| ??- | ~ |

## Multiple Project Files

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

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

**main.c:**

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

**x.c:**

```
#include<x.h>
```

**y.c:**

```
#include<y.h>
```

**z.c:**

```
#include<z.h>
```

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

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

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

## Multiple Compilation Units

Multiple Compilation Units are only supported in the IDE compilers, PCW, PCWH, PCHWD and PCDIDE. When using multiple compilation units, care must be given that pre-processor commands that control the compilation are compatible across all units. It is recommended that directives such as #FUSES, #USE and the device header file all put

in an include file included by all units. When a unit is compiled it will output a relocatable object file (*.o) and symbol file (*.osym).

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

## Full Example Program

Here is a sample program with explanation using CCS C to read adc samples over RS232:

```
#include <16F877A.h>                            // Loads chip specific
definitions
#fuses  NOPROTECT                               // Turn off code protection
#use delay(clock=20000000)                      // Specifies clock speed
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7) // Creates RS232 libraries
void main() {
   unsigned int8 i, value, min, max;
   printf("Sampling:");                         // Printf from the RS232 library

   setup_adc_ports(AN0);                       // Make AN0 a analog pin
   setup_adc(ADC_CLOCK_INTERNAL);              // Start up the ADC
   set_adc_channel(0);                         // Set ADC channel to AN0

   do {
      min=255;
      max=0;
      for(i=0; i<=30; ++i) {
         delay_ms(100);                    // delay function from the delay
library
         value = read_adc();               // Built-in A/D read function
         if(value<min)
            min=value;
         if(value>max)
            max=value;
      }

      printf("\r\nMin: %2X  Max: %2X\n\r",min,max);
   } while (TRUE);
}

--------------------------------------------------------------------------------
// This version of the example uses the C++ cout instead of printf
// and it also shows data streaming through the ICD instead of using
// an RS232 port
#include <16F877A.h>                  // Loads chip specific definitions
#fuses  NOPROTECT                     // Turn off code protection
#use delay(clock=20000000)            // Specifies clock speed
#use rs232(ICD)                       // Creates RS232 libraries (using the ICD)
#include <ios.h>
void main() {
   unsigned int8 i, value, min, max;
```

```
   cout << "Sampling:" << endl;

   setup_adc_ports(AN0);               // Make AN0 a analog pin
   setup_adc(ADC_CLOCK_INTERNAL);      // Start up the ADC
   set_adc_channel(0);                 // Set ADC channel to AN0

   do {
      min=255;
      max=0;
      for(i=0; i<=30; ++i) {
         delay_ms(100);               // delay function from the delay library
         value = read_adc();          // Built-in A/D read function
         if(value<min)
            min=value;
         if(value>max)
            max=value;
      }
      cout << hex << "Min: " << min << "  Max: " << max << endl;
   } while (TRUE);
}
```

# STATEMENTS

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

Note: Items in [ ] are optional

## if

**if-else**

The if-else statement is used to make decisions.

The syntax is:

**if** (expr)
  stmt-1;

[**else**
stmt-2;]

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

**else**-if
This is used to make multi-way decisions.
The syntax is:
**if** (expr)
stmt;
**[else if** (expr)
stmt;]
...
**[else**
stmt;]

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

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

Also See: Statements

## while

**U**sed as a loop/iteration statement.

The syntax is:
**while** (expr)
statement

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

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

Also See: Statements

## do-while

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

**do**
  statement
**while** (expr);

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

Also See: Statements , While

## for

Also used as a loop/iteration statement.
The syntax is:
  **for** (expr1;expr2;expr3)
    statement

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

Example:

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

Also See: Statements

## switch

Also a special multi-way decision maker.
The syntax is

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

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

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

Example:

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

Also See: Statements

## return

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

The syntax is:
**return**(expr);

Example:

```
return (5);
```

Also See: Statements

## goto

The goto statement cause an unconditional branch to the label.

The syntax is:
**goto** label;

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

Example:

```
goto loop;
```

Also See: Statements

## label

The label a goto jumps to.
The syntax is:
**label**: stmnt;

Example:
```
loop: i++;
```

Also See: Statements

## break

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

**break**;

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

Example:
```
break;
```

Also See: Statements

## continue

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

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

Example:
```
continue;
```

Also See: Statements

## expr

The syntax is:
expr**;**

Example:
```
i=1;
```

Also See: Statements

## stmt

Zero or more semi-colon separated.
The syntax is:
**{[**stmt**]}**

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

Also See: Statements

# EXPRESSIONS

## Constants

**123** - Decimal

**123L** - Forces type to & long (UL also allowed)

**123LL** - Forces type to & int32;
*[PCD]* **123LL** - Forces type to & 64 for PCD ULL also allowed

**0123** - Octal

**0x123** - Hex

**0b010010** - Binary

**123.456** - Floating Point

**123F** - Floating Point (FL also allowed)

**123.4E-5** - Floating Point in Scientific Notation

**'x'** - Character

**'\010'** - Octal Character

**'\xA5'** - Hex Character

**'\c'** - Special Character. Where c is one of:

\n Line Feed - Same as \x0a
\r Return Feed - Same as \x0d
\t TAB - Same as \x09
\b Backspace - Same as \x08
\f Form Feed - Same as x0c
\a Bell - Same as \x07
\v Vertical Space - Same as \x0b
\? Question Mark - Same as \x3f
\' Single Quote - Same as \x22
\" Double Quote - Same as \x22
\\ A Single Backslash - Same as \x5c

**"abcdef" -** String (null is added to the end)

## Identifiers

**ABCDE** - Up to 32 characters beginning with a non-numeric. Valid characters are A-Z, 0-9 and _ (underscore). By default not case sensitive Use #CASE to turn on.

**ID[X]** - Single Subscript

**ID[X][X]** - Multiple Subscripts

**ID.ID** - Structure or union reference

**ID->ID** - Structure or union reference

## Operators

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

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

See also: Operator Precedence

## Operator Precedence

| PIn Descending Precedence | | | | Associativity |
|---|---|---|---|---|
| (**expr**) | exor++ | expr->expr | expr.expr | Left to Right |
| ++**expr** | expr**++** | **- -**expr | expr **- -** | Left to Right |
| !**expr** | **~**expr | **+**expr | **-**expr | Right to Left |
| (**type**)**expr** | ***** expr | **&**value | **sizeof**(type) | Right to Left |
| **expr*expr** | expr/expr | expr**%**expr | | Left to Right |
| **expr**+**expr** | expr**-**expr | | | Left to Right |
| **expr**<<**expr** | expr**>>**expr | | | Left to Right |
| **expr**<**expr** | expr**<=**expr | expr**>**expr | expr**>=**expr | Left to Right |
| **expr==expr** | expr**!=**expr | | | Left to Right |
| **expr**&**expr** | | | | Left to Right |
| **expr^expr** | | | | Left to Right |
| **expr \| expr** | | | | Left to Right |
| **expr**&& **expr** | | | | Left to Right |
| **expr \|\| expr** | | | | Left to Right |
| **expr** ? **expr**: **expr** | | | | Right to Left |
| **lvalue = expr** | lvalue**+=**expr | lvalue**-=**expr | | Right to Left |
| **lvalue*=expr** | lvalue**/=**expr | lvalue**%=**expr | | Right to Left |
| **lvalue>>=expr** | lvalue**<<=**expr | lvalue**&=**expr | | Right to Left |
| **lvalue^=expr** | lvalue**\|=**expr | | | Right to Left |
| **expr, expr** | | | | Left to Right |

(Operators on the same line are equal in precedence)

# DATA DEFINITIONS

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

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

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

Variables can also be declared along with the definitions of the *special* types.
For example:

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

SEE ALSO:
Type Specifiers/ Basic Types
Type Qualifiers
Enumerated Types
Structures & Unions
typedef
Named Registers

## Basic Types

| Type-Specifier | Size | Unsigned | Signed | Digits |
|---|---|---|---|---|
| int1 | 1 bit number | 0 to 1 | N/A | 1/2 |
| int8 | 8 bit | 0 to 255 | -128 to 127 | 2-3 |

| | number | | | |
|---|---|---|---|---|
| int16 | 16 bit number | 0 to 65535 | -32768 to 32767 | 4-5 |
| int32 | 32 bit number | 0 to 4294967295 | -2147483648 to 2147483647 | 9-10 |
| int48 | 48 bit number | 0 to 281474976710655 | -140737488355328 to 140737488355327 | 14-15 |
| int64 | 64 bit number | N/A | -9223372036854775808 to 9223372036854775807 | 18-19 |
| float32 | 32 bit float | $-1.5 \times 10^{45}$ to $3.4 \times 10^{38}$ | | 7-8 |
| float48 | 48 bit float (higher precision) | -2.9 x 10 to 1.7 x 10 *[PCD]* -2.9 x $10^{39}$ to 1.7 x $10^{38}$ | | 11-12 |
| float64 | 64 bit float | -5.0 x 10 to 1.7 x 10 *[PCD]* -5.0 x $10^{324}$ to 1.7 x $10^{308}$ | | 15-16 |

| C Standard | Default Type | Default Type - PCD |
|---|---|---|
| **short** | int1 | signed int8 |
| **char** | unsigned int8 | signed int8 |
| **int** | int8 | signed int16 |
| **long** | int16 | signed int32 |
| **long long** | int32 | signed int64 |
| **float** | float32 | float32 |
| **double** | N/A | float64 |

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

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

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

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

## Type Qualifiers

**static -** Variable is globally active and initialized to 0. Only accessible from this compilation unit.

**auto -** Variable exists only while the procedure is active. This is the default and AUTO need not be used.

**double -** A reserved word but is not a supported data type.

**extern -** External variable used with multiple compilation units. No storage is allocated. Is used to make otherwise out of scope data accessible. there must be a non-extern definition at the global level in some compilation unit.

r**egister -** Is allowed as a qualifier however, has no effect.

*[PCD]* Is possible a CPU register instead of a RAM location.

_ **fixed(n) -** Creates a fixed point decimal number where *n* is how many decimal places to implement.

**unsigned -** Data is always positive.

**signed -** Data can be negative or positive.

*[PCD]* **This is the default data type if not specified.**

**volatile -** Tells the compiler optimizer that this variable can be changed at any point during execution.

**const -** Data is read-only. Depending on compiler configuration, this qualifier may just make the data read-only -AND/OR- it may place the data into program memory to save space. (see #DEVICE const=)

**rom -** Forces data into program memory. Pointers may be used to this data but they can not be mixed with RAM pointers.

*[PCD]* r**oml -** Same as rom except only the even program memory locations are used.

**void -** Built-in basic type. Type void is used to indicate no specific type in places where a type is required.

**readonly -** Writes to this variable should be dis-allowed.

**_bif -** Used for compiler built in function prototypes on the same line.

**__attribute__ -** Sets various attributes

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

## Enumerated Types

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

| enum | **[id]** | **{ [ id [ = cexpr]] }** |
|---|---|---|
| | | One or more comma separated |

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

For example:

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

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

## Structures and Unions

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

| struct**[*] [id] {** | **type-qualifier [*] id** | **[:bits];** | **} [id]** |
|---|---|---|---|
| | One or more, semi-colon separated | | Zero or more |

For example:

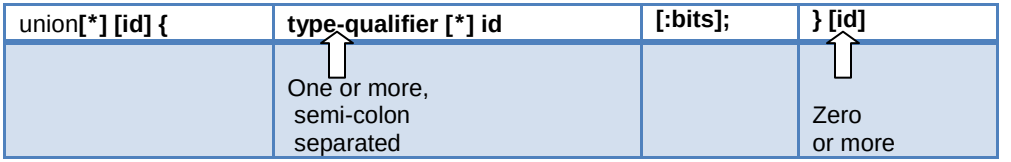
```
struct data_record {
   int   a[2];
   int  b : 2; /*2 bits */
   int    c : 3; /*3 bits*/
   int d;
} data_var;                    //data_record is a structure type
                               //data_var is a variable
```

*[PCD]* **Field Allocation:**

- Fields are allocated in the order they appear.
- The low bits of a byte are filled first.

- Fields 16 bits and up are aligned to a even byte boundary. Some Bits may by unused.
- No Field will span from an odd byte to an even byte unless the field width is a multiple of 16 bits.

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




For example:

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

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

## typedef

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

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

For example:

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

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

## Non-RAM Data Definitions

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

The usage is :

```
addressmod
(name,read_function,write_function,start_address,end_address,
share);
```

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

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

//write procedure for writing n bytes to the memory starting at location addr

    void write_function(int32 addr,int8 *ram, int nbytes){
    }
```

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

**Example:**

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

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

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

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

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

The syntax is :

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

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

For example:

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

## Using Program Memory for Data

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

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

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

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

For example:

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

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

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

The **#org** preprocessor can be used to place the constant to specified address blocks. For example:

The constant ID will be at 1C00.
```
#ORG 0x1C00, 0x1C0F
CONST CHAR ID[10]= {"123456789"};
```
*Note*: Some extra code will precede the 123456789.

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

A special method allows the use of pointers to ROM. This method does not contain extra code at the start of the structure as does constant.
For example:
```
char rom commands[] = {"put|get|status|shutdown"};
```

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

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

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

Where n is optional and id is the table identifier.

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

**#ROM directive:**
Another method is to use #rom to assign data to program memory.

The syntax is:
```
#rom address = {data, data, … , data}
```
For example:

Places 1,2,3,4 to ROM addresses starting at 0x1000
```
#rom 0x1000 = {1, 2, 3, 4}
```
Places null terminated string in ROM
```
#rom 0x1000={"hello"}
```
This method can only be used to initialize the program memory.

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

Writes **data** to program memory
```
write_program_eeprom(address,data);
```

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

Please refer to the help of these functions to get more details on their usage and limitations regarding erase procedures. These functions can be used only on chips that allow writes to program memory. The compiler uses the flash memory erase and write routines to implement the functionality.

The data placed in program memory using the methods listed above can be read from width the following functions:

Reads count bytes from program memory at address to RAM at dataptr.
```
read_program_memory(address, dataptr, count)
```
*[PCD]* Every fourth byte of data is read as 0x00

*[PCD]* Reads count bytes from program memory at the logical address to RAM at dataptr.
```
read_rom_memory((address, dataptr, count)
```
-

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

## Named Registers

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

The syntax is:

register _name type id;

Or

register constant type id;

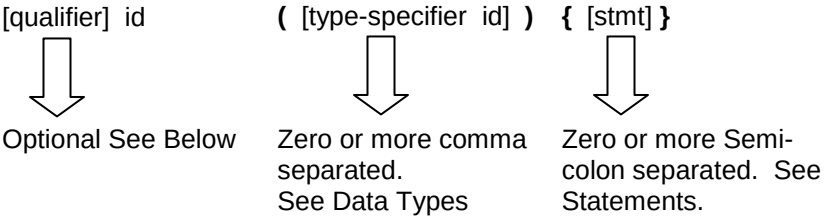
name is a valid SFR name with an underscore before it.

Examples:

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

# FUNCTION DEFINITION

The format of a function definition is as follows:




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.");
```

SEE ALSO:
Overloaded Functions, Reference Parameters, Default Parameters, Variable Parameters

## Overloaded Functions

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

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

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

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

This function finds the square root of a float variable.

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

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

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

## 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);
```

## Default Parameters

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

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

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

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

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

## Variable Argument Lists

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

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

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

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

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

```
    //stop using the list
    va_end(al);
    return(sum);
}
```

Some examples of using this new function:

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

# FUNCTIONAL OVERVIEW

## I2C

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

**Relevant Functions:**

i2c_start() **-** Issues a start command when in the I2C master mode

i2c_write(data) **-** Sends a single byte over the I2C interface

i2c_read() **-** Reads a byte over the I2C interface

i2c_stop() **-** Issues a stop command when in the I2C master mode

i2c_poll() **-** Returns a TRUE if the hardware has received a byte in the buffer

**i2c_transfer(address, wData, wCount, rData, rCount) -** Performs an I2C transfer to and from a device, function does start, restart, write, read, and stop I2C operations; when in I2C master mode.

**i2c_transfer_out(Address, wData, wCount) -** Performs an I2C transfer to a device, function does start, write, and stop I2C operations; when in I2C master mode.

**Relevant Preprocessor:**

#USE I2C - Configures the compiler to support I2C™ to your specifications

**Relevant Interrupts:**

**#INT_SSP -** I2C or SPI activity

**#INT_BUSCOL -** Bus Collision

**#INT_I2C** - I2C Interrupt (Only on 14000)

**#INT_BUSCOL2 -** Bus Collision (Only supported on some PIC18's)

#**INT_SSP2 -** I2C or SPI activity (Only supported on some PIC18's)

*[PCD]* **#INT_mi2c -** Interrupts on activity from the master I2C module

*[PCD]* **#INT_si2c -** Interrupts on activity form the slave I2C module

**Relevant Include Files:**
None - All functions built-in

**Relevant getenv() Parameters:**
**I2C_SLAVE -** Returns a 1 if the device has I2C slave H/W

**I2C_MASTER -** Returns a 1 if the device has a I2C master H/W

**Example Code:**

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

## ADC

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

**Relevant Functions:**
setup_adc(mode) **-** Sets up the a/d mode like off, the adc clock etc.

setup_adc_ports(value) **-** Sets the available adc pins to be analog or digital.

set_adc_channel(channel) **-** Specifies the channel to be use for the a/d call.

read_adc(mode) **-** Starts the conversion and reads the value. The mode can also control the functionality.

adc_done() **-** Returns 1 if the ADC module has finished its conversion.

*[PCD]* **setup_adc2(mode)** - Sets up the ADC2 module, for example the ADC clock and ADC sample time.

[PCD] **setup_adc_ports2(ports, reference)** - Sets the available ADC2 pins to be analog or digital, and sets the voltage reference for ADC2.

[PCD] **set_adc_channel2(channel)** - Specifies the channel to use for the ADC2 input.

[PCD] read_adc2(mode) **-** Starts the sample and conversion sequence and reads the value The mode can also control the functionality.

[PCD] adc_done() **-** Returns 1 if the ADC module has finished its conversion.

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

**Relevant Interrupts:**
**INT_AD** - Interrupt fires when A/D conversion is complete.

**INT_ADOF** - Interrupt fires when A/D conversion has timed out.

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

**Relevant getenv() Parameters:**
**ADC_CHANNELS** - Number of A/D channels.

**ADC_RESOLUTION** - Number of bits returned by read_adc

**Example Code:**

```
#DEVICE ADC=10
...
long value;
...
|
...
setup_adc(ADC_CLOCK_INTERNAL);            // enables the a/d module and
sets the clock to
                                          //  internal adc clock
setup_adc_ports(ALL_ANALOG);              // sets all the adc pins to
analog

set_adc_channel(0);                       // the next read_adc call will
read channel 0
delay_us(10);                            // a small delay is required
after setting channel
                                          // and before read
value=read_adc();                        // starts the conversion and
reads the result and
```

```
read_adc(ADC_START_ONLY);                    // only starts the conversion

value=read_adc(ADC_READ_ONLY);               // reads the result of the last
conversion                                   // and store it in value.

Assuming the device had                      // a10bit  ADC module, value

will range between                           // 0-3FF. If #DEVICE ADC=8 had

been used instead                            // the result will yield 0-FF.

 If #DEVICE ADC=16                           // had been used instead the

result will yield                            // 0-FFC0
```

## Analog Comparator

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

**Relevant Functions:**

**setup_comparator() -** Enables and sets up the analog comparator module. The options vary depending on the device; refer to the device's header file for details.

*[PCD]* **setup_comparator_filter() -** Enables and sets up the analog compartor's digital filter. The options vary depending on the device; refer to the device's header file for details. Not all devices have a digital filter; refer to the device's header file to determine if available.

*[PCD]* **setup_comparator_mask() -** Enables and sets up the analog comparator's output blanking function. The options vary depending on the device; refer to the device's header file for details. Not all devices have an output blanking function; refer to the device's header file to determine if available.

*[PCD]* **setup_comparator_dac() -** Enables and sets up the the common settings analog comparator/DAC modules. The options vary depending on the device. Refer to the device's header file for details. Not all devices have this function. Refer to the device's header file to determine if available.

*[PCD]* **setup_comparator_slope()** - Sets up the analog comparator/DAC slope compensation settings. The options vary depending on the device. Refer to the device's header file for details. Not all devices have this function. Refer to the device's header file to determine if available.

**Relevant Preprocessor:**

None

**Relevant Interrupts:**
**INT_COMP** - Interrupt fires on a comparator change of state.

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

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

**Example Code:**

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

```
[PCD]
setup_comparator(1, CXINB_CXINA);
if(C1OUT)
output_low(PIN_D0);
else
output_high(PIN_D1);
```

## CAN Bus

These functions allow easy access to the Controller Area Network (CAN) features included with the MCP2515 CAN interface chip and the PIC18 MCU. These functions will only work with the MCP2515 CAN interface chip and PIC microcontroller units containing either a CAN or an ECAN module. Some functions are only available for the ECAN module and are specified by the work **(ECAN)** at the end of the description. The listed interrupts are no available to the MCP2515 interface chip.

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

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

**can_set_baud(void);** - Initializes the baud rate of the CAN bus to125kHz, if using a 20 MHz clock and the default CAN-BRG defines, it is called inside the can_init() function so there is no need to call it.

**can_set_mode(CAN_OP_MODE mode);** - Allows the mode of the CAN module to be changed to configuration mode, listen mode, loop back mode, disabled mode, or normal mode.

**can_set_functional_mode (CAN_FUN_OP_MODE mode);** - Allows the functional mode of ECAN modules to be changed to legacy mode, enhanced legacy mode, or first in firstout (fifo) mode. **(ECAN)**

**can_set_id(int* addr, int32 id, int1 ext);** - Can be used to set the filter and mask ID's to the value specified by addr. It is also used to set the ID of the message to be sent.

*[PCD]* **can_set_id(int16 *addr, int32 id, int1 ext) - Can be used to set the filter and mask ID's to the value specified by addr. It is also used to set the ID of the message to be sent on CAN chips.**

*[PCD]* **can_set_buffer_id(BUFFER buffer,int32 id,int1 ext) - Can be used to set the ID of the message to be sent for ECAN devices. (ECAN)**

*[PCD]* **can_get_id(BUFFER buffer,int1 ext) - Returns the ID of a received message.**

c**an_get_id(int * addr, int1 ext); -** Returns the ID of a received message.

**can_putd (int32 id, int * data, int len, int priority, int1 ext, int1 rtr); -** Constructs a CAN packet using the given arguments and places it in one of the available transmit buffers.

*[PCD]* **can_putd(int32 id, int8 *data, int8 &len, struct rx_stat &stat) -** Contructs a CAN packet using the given arguments and places it in one of the available transmit buffers.

**can_getd (int32 & id, int * data, int & len, struct rx_stat & stat); -** Retrieves a received message from one of the CAN buffers and stores the relevant data in the referenced function parameters.

*[PCD]* **can_getd(int32 id, int8 *data, int8 &len, struct rx_stat &stat) -** Retrieves a received message from one of the CAN buffers and stores the relevant data in the referenced function parameters.

c**an_enable_rtr(PROG_BUFFER b); -** Enables the automatic response feature which automatically sends a user created packet when a specified ID is received. **(ECAN)**

c**an_disable_rtr(PROG_BUFFER b); -** Disables the automatic response feature. **(ECAN)**

*[PCD]* **can_kbhit() -** Returns a TRUE if valid CAN messages are available to be retrieved from one of he receive buffers.

c**an_load_rtr (PROG_BUFFER b, int * data, int len); -** Creates and loads the packet that will automatically transmitted when the triggering ID is received. **(ECAN)**

c**an_enable_filter(long filter); -** Enables one of the extra filters included in the ECAN module. **(ECAN)**

c**an_disable_filter(long filter); -** Disables one of the extra filters included in the ECAN module. **(ECAN)**

**can_associate_filter_to_buffer(CAN_FILTER_ASSOCIATION_BUFFERS buffer,CAN_FILTER_ASSOCIATION filter); -** Used to associate a filter to a specific buffer. This allows only specific buffers to be filtered and is available in the ECAN module. **(ECAN)**

**can_associate_filter_to_mask(CAN_MASK_FILTER_ASSOCIATE mask,CAN_FILTER_ASSOCIATION filter); -** Used to associate a mask to a specific buffer. This allows only specific buffer to have this mask applied. This feature is available in the ECAN module.

**can_fifo_getd(int32 &id,int * data,int &len,struct rx_stat & stat); -** Retrieves the next buffer in the fifo buffer. Only available in the ECON module while operating in fifo mode. **(ECAN)**

*[PCD]* **can_fifo_getd(int32 &id,,int8 * data,int8 &len, rx_stat & stat); -** Retrieves the next buffer in the fifo buffer. Only available in the ECON module while operating in fifo mode. **(ECAN)**

*[PCD]* **can_tbe() -** Returns TRUE if a transmit buffer is available to send more data.

*[PCD]* **can_abort()** - Aborts all pending transmissions.

*[PCD]* **can_enable_b_transfer(BUFFER b)** - Sets the specified programmable buffer to be a transmit buffer. **(ECAN)**

*[PCD]* **can_enable_b_receiver(BUFFER b)** - Sets the specified programmable buffer to be a receive buffer. By default, all programmable buffers are set to be receive buffers. **(ECAN)**

*[PCD]* **can_enable_rtr(BUFFER b)** - Enables the automatic response feature. **(ECAN)**

*[PCD]* **can_disable_rtr(BUFFER b)** - Disables the automatic response feature. **(ECAN)**

*[PCD]* **can_load_rtr(BUFFER b, int8 *data, int8 len)** - Creates and loads the packet that will automatically be transmitted when the triggering ID is received. **(ECAN)**

*[PCD]* **can_set_buffer_size(int8 size)** - Set the number of buffers to use. Size can be 4, 6, 8, 12, 16, 24 and 32. By default can_init() sets size to 32. **(ECAN)**

*[PCD]* **can_enable_filter(CAN_FILTER_CONTROLfilter)** - Enables one of the acceptance filters included in the ECAN module. **(ECAN)**

*[PCD]* **can_disable_filter(CAN_FILTER_CONTROLfilter) -** Disables one of the acceptance filters included in the ECAN module. **(ECAN)**

*[PCD]* **can_trb0_putd(int32 id, int8 *data, int8 len, int8 pri, int1 ext, int rtr) -** Contructs a CAN packet using the given arguments and places it in transmit buffer 0. Similar functions available for all transmit buffers 0-7. Buffer must be made a transmit buffer with can_enable_b_transfer() function before function can be use. **(ECAN)**

*[PCD]* **can_enable_interrupts(INTERRUPT setting) -** Enables specified interrupt conditions that cause the #INT_CAN1 interrupt to be tirggered. Available options:

TB - Transmit Buffer interrupt **(ECAN)**
RB - Receive Buffer interrupt **(ECAN)**
RXOV **-** Receive Buffer Overflow interrupt **(ECAN)**
FIFO - FIFO Almost Full interrupt **(ECAN)**
ERR **-** Error interrupt **(ECAN)**
WAK - Wake-Up interrupt **(ECAN)**
IVR - Invalid Message Received interrupt **(ECAN)**
RX0 - Receive Buffer 0 interrupt
RX1 - Receive Buffer 1 interrupt
TX0 - Transmit Buffer 0 interrupt
TX1 - Transmit Buffer 1 interrupt
TX2 - Transmit Buffer 2 interrupt

*[PCD]* **can_disable_interrupts(INTERRUPT setting) -** Disable specified interrupt conditions so they do not cause the #INT_CAN1 interrupt to be triggered. Available options are the same as for the can_enable_interrupts() function. By default, all conditions are disabled.

*[PCD]* **can_config_DMA(void) -** Configures the DMA buffers to use with the ECAN module. It is called inside the can_init() function so there is no need to call it. **(ECAN)**

For PIC microcontrollers that have two CAN or ECAN modules, all the above functions are available for the second module, and they begin with can2 instead of can.
can2_init(); or can2_kbhit();

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
**#int_canirx -** This interrupt is triggered when an invalid packet is received on the CAN.

**#int_canwake -** This interrupt is triggered when the PIC is woken up by activity on the CAN.

#**int_canerr -** This interrupt is triggered when there is an error in the CAN module.

#**int_cantx0 -** This interrupt is triggered when transmission from buffer 0 has completed.

#**int_cantx1 -** This interrupt is triggered when transmission from buffer 1 has completed.

#**int_cantx2 -** This interrupt is triggered when transmission from buffer 2 has completed.

#**int_canrx0 -** This interrupt is triggered when a message is received in buffer 0.

#**int_canrx1 -** This interrupt is triggered when a message is received in buffer 1.

*[PCD]* **#int_can1 -** Interrupt for CAN or ECAN module 1. This interrupt is triggered when one of the conditions set by can_enable_interrupts() is met.

*[PCD]* **#int_can2 -** Interrupt for CAN or ECAN moduel 2. This interrupt is triggered when one of the conditions set by the can2_enable_interrupts() is met. This interrupt is only available on devices that have two CAN or ECAN modules.

**Relevant Include Files:**
**can-mcp2510.c** - Drivers for the MCP2510 and MCP2515 interface devices.

**can-18xxx8.c -** Drivers for the built-in CAN module.

c**an-18F4580.c -** Drivers for the built-in ECAN module.

*[PCD]* **can-dsPIC30.c** - Drivers for the built-in CAN module on dsPIC30F devices.

*[PCD]* **can-PIC24.c** - Drivers for the built-in ECAN mdoule on PIC24HF and dsPIC33FJ devices.

**Relevant getenv() Parameters:**
None

**Example Code:**
```
  can_init();                              // initializes the CAN bus
  can_putd(0x300,data,8,3,TRUE,FALSE);   // places a message on the CAN
  bus with ID=0x300
                                          // and eight bytes of data pointed
to by "data",
```

```
                                        // the TRUE create an extended ID,
the FALSE
                                        // creates
  can_getd(ID,data,len,stat);           // retrieves a message from the
  CAN bus storing the
                                       // ID in the ID variable, the data
in the array
                                       // pointed to by "data", the number
of data bytes
                                       // in len, and statistics about the
data in
                                       // the stat structure.
```

## CCP

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

**Relevant Functions:**
**setup_ccp1(mode)** - Sets the mode to capture, compare or PWM.

**set_pwm1_duty(value)** - The value is written to the pwm1 to set the duty.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_CCP1 - Interrupt fires when capture or compare on CCP1.

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

**Relevant getenv() Parameters:**
CCP1 - Returns 1 if the device has CCP1

**Example Code:**

```
  #int_ccp1
  void isr()
  {
    rise=CCP_1;                 // CCP_1 is the time the pulse went high
```

```
  fall=CCP2;                  // CCP_2 is the time the pulse went low
  pulse_width=fall-rise;      // pulse width
}
...
setup_ccp1(CCP_CAPTURE_RE);    // Configure CCP1 to capture rise
setup_ccp2(CCP_CAPTURE_FE);    // Configure CCP2 to capture fall
setup_timer_1(T1_INTERNAL);    // Start timer 1
```

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

## Code Profile

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

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

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

**Relevant Functions:**
**profileout() -** Send a user specified message or variable to be displayed or logged by the code profile tool.

**Relevant Preprocessor:**
**#use profile() -** Global configuration of the code profile run-time on the microcontroller.

#profile **-** Dynamically enable/disable specific elements of the profiler.

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

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

**Relevant getenv() Parameters:**
None

**Example Code:**

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

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

## Configuration Memory

The Configuration Memory is readable and writable on all PIC18, PIC24, dsPIC30 and dsPIC33 devices. Enhanced 16 devices have the configuration memory that is readable and the user ID is readable and writable..

*[PCD]* The Configuration Memory contains the configuration bits for items such as the oscillator mode, watchdog timer enable, etc. These configuration bits are set by the CCS C Compiler usually through a #fuse. CCS provides an API that allows for these bits to be changed in run-time.

**Relevant Functions:**

**write_configuration_memory(ramaddress, count) -** Writes count bytes, no erase needed.

**write_configuration_memory(offset,ramaddress, count) -** Writes count bytes, no erase needed starting at byte address offset.

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

*[PCD]* **write_configuration_memory(offset, ramPtr, n);** - Read n bytes of configuration memory, save to ramPtr.

**read_configuration_memory(ramaddress,count) -** Read count bytes of configuration memory.

*[PCD]* **read_configuration_memory(ramPtr, n);** - Read n bytes of configuration memory is set through a #FUSE.

**read_device_info()** - Read count bytes from Device Information Area memory.

**read_config_info()** - Read count bytes from Device Configuration Information memory.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
None

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

**Relevant getenv() Parameters:**
None

**Example Code:**

```
#int16 data=0xc32;
...
write_configuration_memory(data,2); // writes 2 bytes to the config
memory
```

## CRC

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

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

**Relevant Functions:**
**setup_crc(polynomial)** - This will setup the CRC polynomial.

**crc_init(data)** - Sets the initial value used by the CRC module.

**crc_calc(data)** - Returns the calculated CRC value.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_CRC - On completion of CRC calculation.

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

**Relevant getenv() Parameters:**
None

**Example Code:**

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

setup_crc(15, 3, 1);          //CRC Polynomial is X16+X15+X3+X1 +1
                             //or polynomial=8005h
crc_init(0xFEEE);             //Starts the CRC accumulator outo f 0xFEEE
result=crc_calc(&data[0],8): //Calculates the CRC
```

## DAC

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

**Relevant Functions:**
setup_dac(divisor) **-** Sets up the DAC e.g. Reference voltages.

dac_write(value) **-** Writes the 8-bit value to the DAC module.

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

*[PCD]* dac_write(channel, value) **-** Writes the 16-bit value to the specified channel.

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

**Relevant Interrupts:**
None

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

**Relevant getenv() Parameters:**
None

**Example Code:**

```
  int8 i=0;
  setup_dac (DAC_VSS_VDD);
  while (TRUE) {
  itt;
  dac_write(i);
  }
```

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

## Data Eeprom

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

**Relevant Functions:**
read_eeprom(address) **-** Reads the data EEPROM memory location

write_eeprom(address, value) **-** Erases and write value to data EEPROM location address. Except for PCB devices with EEPROM, such as PIC12F519; it only writes the value.

erase_eeprom(address) **-** Erases a row of the EEPROM of Flash memory. Only available on PCB devices with EEPROM, such as PIC12F599.

read_eeprom(address, [N]) **-** Reads N bytes of data EEPROM starting at memory location address. The maximum return size is int64.

read_eeprom(address, [variable]) **-** Reads from EEPROM to fill variable starting at address.

**read_eeprom(address, pointer, N) -** Reads N bytes, starting at address, to pointer.

**write_eeprom(address, value) -** Writes value to EEPROM address.

**write_eeprom(address, pointer, N) - Writes N bytes to address from pointer**

**Relevant Preprocessor:**
#ROM address={list} **-** Can also be used to put data EEPROM memory data into the hex file.

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

**Relevant Interrupts:**
**INT_EEPROM -** Interrupt fires when EEPROM write is complete.

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

**Relevant getevn() Parameters:**
**DATA_EEPROM -** Size of data EEPROM memory.

**Example Code:**

For 18F452

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

  write_eeprom(0x0,0x12);        //write 0x12 to data eeprom location 0
       value-read_eeprom(0x)     // reads data eeprom location 0x0
  returns 0x12

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

## DCI

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

**Relevant Functions:**
setup_dci(configuration, data size, rx config, tx config, sample rate);**-** Initializes the DCI module.

**setup_adc_ports(value)** - Sets the available ADC pins to be analog or digital.

**set_adc_channel(channel)** - Specifies the channelt o be used for the A/D call.

**read_adc(mode) -** Starts the conversion and reads the value. The mode can also control the functionality.

adc_done() **-** Returns 1 if the ADC module has finished its conversion.

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

**Relevant Interrupts:**
INT_DCI - Interrupt fires on a number (user configurable) of data words received.

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

**Relevant getenv() Parameters:**
None

**Example Code:**
```
signed int16 left_channel, right_channel;

dci_initializes((I2S_MODE|DCI_MASTER|DCI_CLOCK_OUT|
SAMPLE_RISING_EDGE|UNDERFLOW_LAST|MULTI_DEVICE_BUS),DCI_1WORD_FRAME|
DCI_16BIT_WORD|DCI_2WORD_INTERRUPT, RECEIVE_SLOT0|RECEIVE_SLOT1,
TRANSMIT_SLOT0|TRANSMIT_SLOT1, 6000);
...
dci_start();
...
while(1)
{
```

```
    dci_read(&left_channel, &right_channel);
    dci_write(&left_channel, &right_channel);
  }
```

## DMA

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

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

**Relevant Functions:**

**setup_dma(channel, peripheral,mode) -** Configures the DMA module to copy data from the specified peripheral to RAM allocated for the DMA channel.

**dma_start(channel, mode,address) -** Starts the DMA transfer for the specified channel in the specified mode of operation.

**dma_status(channel) -** This function will return the status of the specified channel in the DMA module.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
#INT_DMAX - Interrupt on channel X after DMA block or half block transfer.

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

**Relevant getenv() Parameters:**
None

**Example Code:**

```
setup_dma(1,DMA_IN_SIP1,DMA_BYTE);  // Setup channel 1 of the DMA
module to
                                    // read the SPI1 channel in byte
mode.
dma_start(1,DMA_CONTINUOUS|DMA_PING_PONG, 0x2000);
                                    // Start the DMA channel with the
DMA
                                    // RAM address of 0x2000
```

## Data Signal Modulator

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

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

**Relevant Functions:** ***(8 bit or 16 bit depending on the device)***
**setup_dsm(mode,source,carrier) -** Configures the DSM module and selects the source signal and carrier signals.

s**etup_dsm(TRUE) -** Enables the DSM module.

**setup_dsm(FALSE) -** Disables the DSM module.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
None

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

**Relevant getenv() Parameters:**
None

**Example Code:**

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

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

## Extended RAM

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

**Relevant Functions:**
**write_extended_ram(p,addr,ptr,n); -** Writes n bytes from ptr to extended RAM page p starting at address addr.

r**ead_extended_ram(p,addr,ptr,n); -** Reads n bytes from extended RAM page p starting a address addr to ptr.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
None

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

**Relevant getenv() Parameters:**
None

**Example Code:**

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

## External Memory

Some PIC18 devices have the external memory functionality where the external memory can be mapped to external memory devices like (Flash, EPROM or RAM). These functions are available only on devices that support external memory bus.

## General Purpose I/O

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

**Relevant Functions:**

**output_high(pin) -** Sets the given pin to high state.

o**utput_low(pin) -** Sets the given pin to the ground state.

o**utput_float(pin) -** 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.

o**utput_x(value) -** Outputs an entire byte to the port.

o**utput_bit(pin,value) -** Outputs the specified value (0,1) to the specified I/O pin.

i**nput(pin) -** The function returns the state of the indicated pin.

i**nput_state(pin) -** This function reads the level of a pin without changing the direction of the pin as INPUT() does.

s**et_tris_x(value) -** Sets the value of the I/O port direction register. A '1' is an input and '0' is for output.

i**nput_change_x( ) -** This function reads the levels of the pins on the port, and compares them to the last time they were read to see if there was a change, 1 if there was, 0 if there was not.

s**et_open_drain_x(value) -** This function sets the value of the I/O port Open-Drain register. A | makes the output open-drain and 0 makes the output push-pull.

s**et_input_level_x(value) -** This function sets the value of the I/O port Input Level Register. A 1 sets the input level to ST and 0 sets the input level to TTL.

*[PCD]* s**et_open_drain_x() -** Sets the value of the I/O port Open-Drain Control register. A '1' sets it as an open-drain output, and a '0' sets it as a digital output.

**Relevant Preprocessor:**

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

#USE FAST_IO(*port*) - This directive will configure the I/O port to use the fast method of performing I/O. The user will be responsible for setting the port direction register using the set_tris_x() function.

#USE FIXED_IO (*port_outputs=;in,pin?*) - This directive set particular pins to be used an input or output, and the compiler will perform this setup every time this pin is used.

**Relevant Interrupts:**

None

**Relevant Include Files:**

None, all functions are built-in

**Relevant getenv() Parameters:**

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

**Example Code:**

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

## Input Capture

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

**Relevant Functions:**

setup_capture(x, mode) **-** Sets the operation mode of the input capture module x

**get_capture(x, wait) -** Reads the capture event time from the ICxBUF result register. If wait is true, program flow waits until a new result is present. Otherwise the oldest value in the buffer is returned.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_ICx - Interrupt fires on capture event as configured

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

**Relevant getenv() Parameters:**
None

**Example Code:**

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

## Internal LCD

Some families of PIC microcontrollers can drive a glass segment LCD directly, without the need of an LCD controller. For example, the PIC16C92X, PIC16F91X, and PIC16F193X series of chips have an internal LCD driver module.

**Relevant Functions:**

**setup_lcd(mode, prescale, [segments]) -** Configures the LCD Driver Module to use the specified mode, timer prescaler, and segments. For more information on valid modes and settings, see the setup_lcd( ) manual page and the *.h header file for the PIC micro-controller being used.

**lcd_symbol(symbol, segment_b7 ... segment_b0) -** The specified symbol is placed on the desired segments, where segment_b7 to segment_b0 represent SEGXX pins on the PIC micro-controller. For example, if bit 0 of symbol is set, then segment_b0 is set, and if segment_b0 is 15, then SEG15 would be set.

**lcd_load(ptr, offset, length) -** Writes length bytes of data from pointer directly to the LCD segment memory, starting with offset.

**lcd_contrast (contrast) -** Passing a value of 0 – 7 will change the contrast of the LCD segments, 0 being the minimum, 7 being the maximum.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
#INT_LCD - LCD frame is complete, all pixels displayed

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

**Relevant getenv() Parameters:**
LCD - Returns TRUE if the device has an Internal LCD Driver Module.

**Example Code:**

```
                                                // How each segment of
the LCD is set
                                               //(on or off) for the
ASCII digits to 9.
byte CONST DIGIT_MAP[10]={0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE,
0xE0, 0xFE, 0xE6};

                                                // Define the segment
information for the
                                               //first digit of the LCD
#define DIGIT1   COM1+20, COM1+18, COM2+18, COM3+20, COM2+28, COM1+28,
COM2+20, COM3+18
                                                // Displays the digits 0
to 9 on the first
                                               //digit of the LCD.

for(i = 0; i <= 9; i++) {
   lcd_symbol( DIGIT_MAP[i], DIGIT1 );
   delay_ms( 1000 );
}
```

## Internal Oscillator

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

to a higher/lower value in software. They are also software tunable. Some chips also provide the PLL option for the internal oscillator.

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

**Relevant Functions:**

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

**setup_oscillator()** - Explicitly configures the oscillator.

**Relevant Preprocessor:**

*[PCD] C*#FUSES - Specifies the values loaded in the device configuration memory. May be used to setup the oscillator configuration.

**Relevant Interrupts:**

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

*[PCD]* #INT_OSCFAIL - Interrups on oscillator failure

**Relevant Include Files:**

None, all functions are built-in

**Relevant getenv() Parameters:**

*[PCD]* CLOCK - Returns the clock speed specified by #use delay()

*[PCD]* FUSE_SETxxx - Returns 1 if the fuse xxxx is set.

**Example Code:**

**For PIC18F8722**

```
setup_oscillator(OSC_32MHZ); //sets the internal oscillator to 32Mhz
(PLL enabled)
```

If the internal oscillator fuse option are specified in the #fuses and a valid clock is specified in the #use delay(clock=xxx) directive the compiler automatically sets up the oscillator. The #use delay statements should be used to tell the compiler about the oscillator speed.

## Interrupts

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

**Relevant Functions:**
**disable_interrupts() -** Disables the specified interrupt.
e**nable_interrupts() -** Enables the specified interrupt.

e**xt_int_edge() -** Enables the edge on which the edge interrupt should trigger. This can be either rising or falling edge.

c**lear_interrupt() -** This function will clear the specified interrupt flag. This can be used if a global isr is used, or to prevent an interrupt from being serviced.

i**nterrupt_active() -** This function checks the interrupt flag of specified interrupt and returns true if flag is set.

i**nterrupt_enabled() -** This function checks the interrupt enable flag of the specified interrupt and returns TRUE if set.

**Relevant Preprocessor:**
#DEVICE HIGH_INTS= - This directive tells the compiler to generate code for high priority interrupts.

#INT_XXX fast - This directive tells the compiler that the specified interrupt should be treated as a high priority interrupt.

*[PCD]* #INT_XXX level=x - x is an int 0-7, that selects the interrupt priority level for that interrupt.

*[PCD]* #INT_XXX fast - This directive makes use of shadow registers for fast register save. This directive can only be used in one ISR

**Relevant Interrupts:**
#int_default - This directive specifies that the following function should be called if an interrupt is triggered but no routine is associated with that interrupt.
#int_global - This directive specifies that the following function should be called whenever an interrupt is triggered. This function will replace the compiler generated interrupt dispatcher.
#int_xxx - This directive specifies that the following function should be called whenever the xxx interrupt is triggered. If the compiler generated interrupt dispatcher is used, the compiler will take care of clearing the interrupt flag bits.

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

**Relevant getenv() Parameters:**
None

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

## Low Voltage Detect

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

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

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_LOWVOLT - Interrup fires on low voltage detect

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

**Relevant getenv() Parameters:**
None

**Example Code:**
For PIC18F8722
```
  setup_low_volt_detect(LVD_36|LVD_TRIGGER_ABOVE);
                                      // sets the trigger level as 3.6
  volts and
                                       // trigger direction as above. The
  interrupt
```

```
                                        // if enabled is fired when the
voltage is
                                        // above 3.6 volts.
```

## Output Compare/PWM Overview

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

**Relevant Functions:**
**setup_comparex (*x, mode*) -** Sets the ***mode*** of the output compare / PWM module x

s**et_comparex_time ( *x*, *ocr*, [*ocrs*]) -** Sets the OCR and optionally OCRS register values of module ***x***.

s**et_pwm_duty (*x*, *value*) -** Sets the PWM duty cycle of module ***x*** to the specified ***value***

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_OCx - Interrup fires after a compare event has occurred.

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

**Relevant getenv() Parameters:**
None

**Example Code:**

```
                                        //Outputs a 1 second pulse on the
OC2 pin
                                       //using dual compare mode on a PIC
with
                                       //an instruction clock of (20Mhz/4)
int16 OCR_2=0x1000;                     //Start pulse when timer is at
0x1000
```

```
int15 OCRS_2=0x5C4B;                //End pulse after 0x04C4B timer
counts
                                    //(0x1000+0x04C4B
                                    //(1sec)/[(4/20000000*256]=0x04C4B
                                    //256-timer prescaler value (set in
code)
set_compare_time(2, OCR_2,OCRS_2);
setup_compare(2, COMPARE_SINGLE_PULSE|COMPARE_TIMER3);
setup_timer3(TMR_INTERNAL|TMR_DIV_BY_256);
```

## Motor Control PWM

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

**Relevant Functions:**

**setup_motor_pwm(pwm,options, timebase); -** Configures the motor control PWM module.

s**et_motor_pwm_duty(pwm,unit,time) -** Configures the motor control PWM unit duty.

s**et_motor_pwm_event(pwm,time) -** Configures the PWM event on the motor control unit.

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

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

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
#INT_PWM1 - PWM Timebase Interrupt

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

**Relevant getenv() Parameters:**
None

**Example Code:**

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

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

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

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

## PMP/EPMP

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

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

**Relevant Functions:**

**setup_psp (options,address_mask) -** This will setup the PSP module for various mode and specifies which address lines to be used.

**setup_pmp_csx(options,[offset]) -** Sets up the Chip Select X Configuration, Mode and Base Address registers.

*[PCD]* setup_pmp **(options,address_mask) -** This will setup the PMP/EPMP module for various mode and specifies which address lines to be used.

**setup_psp_cs(options) -** Sets up the Chip Select X Configuration and Mode registers.

**psp_output_full() -** This will return the status of the output buffers.

*[PCD]* **pmp_address(address) -** Configures the address register of the PMP module with the destination address during Master mode operation.

[PCD] **pmp_input_full () -** This will return the status of the input buffers.

[PCD] **psp_input_full()** - This will return the status of the input buffers.

[PCD] **pmp_output_full() -** This will return the status of the output buffers.

[PCD] **pmp_overflow () -** This will return the status of the output buffer underflow bit.

[PCD] **pmp_read( ) -** Reads a byte of data.

[PCD] **psp_read(address)/ psp_read() -** psp_read() will read a byte of data from the next buffer location and psp_read ( address ) will read the buffer location address.

[PCD] pmp_write **(data) -** Write the data byte to the next buffer location.

[PCD] **psp_write(address,data)/ psp_write(data) -** This will write a byte of data to the next buffer location or will write a byte to the specified buffer location.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
#INT_PMP - Interrupt on read or write strobe

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

**Relevant getenv() Parameters:**
None

**Example Code:**

```
setup_pmp( PAR_ENABLE |       // Sets up Master mode with
                              // address lines  PMA0:PMA7
PAR_MASTER_MODE_1|
PAR_STOP_IN_IDLE,0x0FF);

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

## Power PWM

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

**Relevant Functions:**
**setup_power_pwm(config) -** Sets up the PWM clock, period, dead time etc.

s**etup_power_pwm_pins(module x) -** Configure the pins of the PWM to be in Complementary, ON or OFF mod.

s**et_power_pwmx_duty(duty) -** Stores the value of the duty cycle in the PDCXL/H register. This duty cycle value is the time for which the PWM is in active state.

s**et_power_pwm_override(pwm,override,value) -** This function determines whether the OVDCONS or the PDC registers determine the PWM output .

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
#INT_PWMTB - PWM Timebase Interrupt (Only available on PIC18XX31)

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

**Relevant getenv() Parameters:**
None

**Example Code:**

```
  ....
  long duty_cycle, period;
  ...
                                  // Configures PWM  pins to be
  ON,OFF
                                  // or in Complimentary mode.

  setup_power_pwm_pins(PWM_COMPLEMENTARY ,PWM_OFF, PWM_OFF,
  PWM_OFF_;
                                  // Sets up PWM clock , postscale
  and
                                  // period. Here period is used
  to set the
                                  // PWM Frequency as follows:
                                  // Frequency=Fosc/(4* (period+1)
                                  // *postscale)

  setup_power_pwm(PWM_CLOCK_DIV_4|PWM_FREE_RUN,1,0,period,0,1,0);
  set_power_pwm0_duty(duty_cycle));     // Sets the duty cycle of
  the PWM 0,1 in
                                        // Complementary mode
```

# Program EEPROM

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

### Relevant Functions:

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

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

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

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

**read_program_memory(address,dataptr,count) -** Read count bytes from program memory at address to dataptr.

**read_calibration_memory(cal_word) -** Read one of the calibration words from calibration memory on MCP191xx devices.

*[PCD]* **read_rom_memory(address,dataptr,count) -** Reads count bytes from program memory from address.

### Relevant Preprocessor:

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

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

### Relevant Interrupts:

**INT_EEPROM -** Interrupts fire when EEPROM write is complete.

### Relevant Include Files:

None, all functions built-in

### Relevant getenv() Parameters:

**PROGRAM_MEMORY** - Size of program memory.

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

**FLASH_WRITE_SIZE** - Smallest number of bytes written in Flash.

**FLASH_ERASE_SIZE** - Smallest number of bytes erased in Flash.

*[PCD]* **MIN_FLASH_WRITE** - Smallest number of bytes that can be written to Flash with write_program_memory() function.

**Example Code:**

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

```
#rom 0xa00={1,2,3,4,5}              //inserts this data into the hex
file.
erase_program_eeprom(0x1000);           //erases 64 bytes starting at
0x1000
write_program_eeprom(0x1000,0x1234);    //writes 0x1234 to 0x1000
value=read_program_eeprom(0x1000);      //reads 0x1000 returns 0x1234
write_program_memory(0x1000,data,8);    //of 64 and writes 8 bytes from
data to 0x1000
read_program_memory(0x1000,value,8);    //reads 8 bytes to value from
0x1000
erase_program_eeprom(0x1000);           //erases 64 bytes starting at
0x1000
write_program_memory(0x1010,data,8);    //writes 8 bytes from data to
0x1000
read_program_memory(0x1000,value,8);   //reads 8 bytes to value from
0x1000
```

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

**WRITE_PROGRAM_EEPROM** - Writes 2 bytes,does not erase (use ERASE_PROGRAM_EEPROM)

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

**ERASE_PROGRAM_EEPROM** - Will erase a block. The lowest address bits are not used.

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

**WRITE_PROGRAM_EEPROM** - Writes 2 bytes, no erase is needed.

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

**ERASE_PROGRAM_EEPROM** - Not available.

*[PCD]*

```
#rom0x1000=(1,2,3,4)                  //Inserts this data into the hex
file
erase_program_memory(0x1000);         //Erases flash page containing
address
                                      //0x1000, erase size depends on
                                      //FLASH_ERASE_SIZE
write_program_memory(0x1000,data,12); //Write 12 bytes from data
program memory
                                      //starting at address 0x1000, if
address
```

```
                          //0x1000 is the start of a flash erase
                           //block, then erase will be done first.
read_program_memory(0x1000,value,12); //Reads 12 bytes to value from
program
                           //memory starting at address 0x1000.
WRITE_PROGRAM_MEMORY       //Writes any number of bytes that is a
                          //multiple of MIN_FLASH_WRITE.  Will
                         //erase a block whenever the first
(lowest)                 //byte in a block is written to. If the
                            //first address is not the start of a block
                           //that block is not erased.
ERASE_PROGRAM_MEMORY       //Erases a block of size FLASH_ERASE_SIZE.
                            //The lowest address bit are not used.
                           //i.e. any  address passed to function will
                           cause block it is contained in to be erased.
```

## PSP

These options let to configure and use the Parallel Slave Port on the supported devices.

**Relevant Functions:**
**setup_psp(mode) -** Enables/disables the psp port on the chip.

p**sp_output_full() -** Returns 1 if the output buffer is full(waiting to be read by the external bus).

p**sp_input_full() -** Returns 1 if the input buffer is full(waiting to read by the cpu).

p**sp_overflow() -** Returns 1 if a write occurred before the previously written byte was read.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_PSP - Interrupt fires when PSP data is in

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

**Relevant getenv() Parameters:**
PSP - Returns 1 if the device has PSP

**Example Code:**

```
while(psp_output_full());     //waits till the output buffer is
cleared
psp_data=command;             //writes to the port
```

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

## QEI

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

**Relevant Functions:**
**setup_qei(options, filter,maxcount) -** Configures the QEI module.

q**ei_status( ) -** Returns the status of the QEI module

q**ei_set_count(value) -** Writes a 16-bit value to the position counter.

q**ei_get_count( ) -** Reads the current 16-bit value of the position counter.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_QEI - Interrupt on rollover or underflow of the position counter

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

**Relevant getenv() Parameters:**
None

**Example Code:**

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

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

## RS232 I/O

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

### Relevant Functions:

**getc() or getch() / getchar() or fgetc() -** Gets a character on the receive pin (from the specified stream in case of fgetc, stdin by default). Use KBHIT to check if the character is available.

g**ets() or fgets() -** Gets a string on the receive pin (from the specified stream in case of fgets, STDIN by default). Use getc to receive each character until return is encountered.

**putc() or putchar() or / fputc() -** Puts a character over the transmit pin (on the specified stream in the case of fputc, stdout by default).

p**uts() or fputs() -** Puts a string over the transmit pin (on the specified stream in the case of fputc, stdout by default). Uses putc to send each character.

p**rintf() or fprintf() -** Prints the formatted string (on the specified stream in the case of fprintf, stdout by default). Refer to the printf help for details on format string.

k**bhit() -** Return true when a character is received in the buffer in case of hardware RS232 or when the first bit is sent on the RCV pin in case of software RS232. Useful for polling without waiting in getc.

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

a**ssert(condition) -** Checks the condition and if false prints the file name and line to STDERR. Will not generate code if #DEFINE NODEBUG is used.

p**error(message) -** Prints the message and the last system error to STDERR.

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

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

t**x_buffer_bytes() -** When using transmit buffer, returns the number of bytes in buffer that still need to be sent.

t**x_buffer_full() -** When using transmit buffer, returns TRUE if transmit buffer is full.

r**eceive_buffer_full() -** When using receive buffer, returns TRUE if receive buffer is full.

t**x_buffer_available() -** When using transmit buffer, returns number of characters that can be put into transmit buffer before it overflows.

#**useRS232 -** Configures the compiler to support RS232 to specifications.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_RDA - Interrupt fires when the receive data available.

INT_TBE - Interrup fires when the transmit data empty.

*Some devices have more than one hardware UART, hence more interrupts.

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

**Relevant getenv() Parameters:**
UART - Returns the number UARTs on this device.

AUART - Returns TRUE if this UART is an advanced UART.

UART_RX - Returns the receive pin for the first UART on this device (see PIN_XX)

UART_TX - Returns the transmit pin for the first UART on this device.

UART2_RX - Returns the receive pin for the second UART on this device.

UART2-TX - Returns the transmit pin for the second UART on this device.

**Example Code:**

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

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

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

## RTCC

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

- Time: Hour, Minute and Seconds.
- 24-hour format (Military Time)
- Calendar: Weekday, Date, Month and Year.

- Alarm Configurable.
- Requirements: External 32.768 kHz Clock Crystal.

**Relevant Functions:**

**setup_rtc (options, calibration); -** This will setup the RTCC module for operation and also allows for calibration setup.

r**tc_write(rtc_time_t datetime) -** Writes the date and time to the RTCC module.

r**tc_read(rtctime_t datetime) -** Reads the current value of Time and Date from the RTCC module.

s**etup_rtc_alarm(options, mask, repeat); -** Configures the alarm of the RTCC module.

r**tc_alarm_write(rtctime_t datetime); -** Writes the date and time to the alarm in the RTCC module.

r**tc_alarm_read(rtctime_t datetime); -** Reads the date and time to the alarm in the RTCC module.

**Relevant Preprocessor:**
None

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

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

**Relevant getenv() Parameters:**
None

**Example Code:**

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

## RTOS

These functions control the operation of the CCS Real Time Operating System (RTOS). This operating system is cooperatively multitasking and allows for tasks to be scheduled

to run at specified time intervals. Because the RTOS does not use interrupts, the user must be careful to make use of the rtos_yield() function in every task so that no one task is allowed to run forever.

**Relevant Functions:**

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

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

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

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

**rtos_msg_poll()** - Returns true if there is data in the task's message queue.

**rtos_msg_read()** - Returns the next byte of data contained in the task's message queue.

**rtos_msg_send(task,byte)** - Sends a byte of data to the specified task. The data is placed in the receiving task's message queue.

**rtos_yield()** - Called with in one of the RTOS tasks and returns control of the program to the rtos_run() function. All tasks should call this function when finished.

**rtos_signal(sem)** - Increments a semaphore which is used to broadcast the availability of a limited resource.

**rtos_wait(sem)** - Waits for the resource associated with the semaphore to become available and then decrements to semaphore to claim the resource.

**rtos_await(expre)** - Will wait for the given expression to evaluate to true before allowing the task to continue.

**rtos_overrun(task)** - Will return true if the given task over ran its allotted time.

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

**Relevant Preprocessor:**

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

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

**#TASK** - Specifies the rate at which the task should be called, the maximum time the task shall be allowed to run, and how large its queue should be.

**Relevant Interrupts:**
None

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

**Relevant getenv() Parameters:**
None

**Example Code:**

```
#USE RTOS(timer=0,minor_cycle=20ms)      // RTOS will use timer zero,
minor cycle
                                        // will be 20ms
...
int sem;
...
#TASK(rate=1s,max=20ms,queue=5)          // Task will run at a rate of
once per second
void task_name();                        // with a  maximum running time
of 20ms and
                                        // a 5 byte queue
rtos_run();                              // begins the RTOS
rtos_terminate();                        // ends the RTOS
rtos_enable(task_name);                  // enables the previously
declared task.
rtos_disable(task_name);                 // disables the previously
declared task
rtos_msg_send(task_name,5);              // places the value 5 in
task_names queue.
rtos_yield();                            // yields control to the RTOS
rtos_signal(sem);                         // signals that the resource
represented by
                                        // sem is available.
```

**For more information on the CCS RTOS please**

## SPI

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

**Relevant Functions:**

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

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

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

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

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

**spi_set_txcnt(value)** - Sets the number of SPI transfers to drive SS1 pin to active level. Only available on PIC18 devices with a dedicated SPI peripheral.

**Relevant Preprocessor:**

None

**Relevant Interrupts:**

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

*[PCD]* **#int_spi1** - Interrupts on the activity from the first SPI module.

*[PCD]* **#int_spi2** - Interrupts on the activity from the second SPI module.

**Relevant Include Files:**

None, all functions built-in to the compiler.

**Relevant getenv() Parameters:**

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

**Example Code:**

```
                                    //configure the device to be a
master,
                                    //data transmitted on H-to-L clock
transition
setup_spi(SPI_MASTER|SPI_H_TO_L|SPI_CLK_DIV_16);
spi_write(0x80);                            //write 0x80 to SPI device
value=spi_read();                           //read a value from the
SPI device
```

```
value=spi_read(0x80);                    //write 0x80 to SPI device
the same
                                         //time reading a value.
spi_set_txcnt(3);                        //drives SS1 pin to active
level
                                         //for 3 SPI transfers
```

## Timers

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

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

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

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

**Relevant Functions:**

**setup_timer_X() -** Configures the timer peripheral. X may be any valid timer for the target device. Consult the target datasheet or use getenv to find the valid timers.

g**et_timerX() -** Retrieves the current 16-bit value of the timer.

g**et_timerXY() -** Gets the 32-bit value of the concatenated timers X and Y (where XY may only be 23, 45, 67, 89).

s**et_timerX() -** Sets the value of timerX.

s**et_timerXY() -** Sets the 32-bit value of the concatenated timers X and Y (where XY may only be 23, 45, 67, 89).

**Relevant Preprocessor:**

None

**Relevant Interrupts:**
#int_timerX - Interrupts on timer overflow (period match). X is any valid timer number.
*When using a 32-bit timer, the odd numbered timer-interrupt of the hybrid timer must be used (i.e. when using 32-bit Timer 23, #int_timer3).

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

**Relevant getenv() Parameters:**
TIMERX - Returns 1 if the device has the timer peripheral X. X may be 1-9.

**Example Code:**
```
/*Setup timer1 as an external realtime clock that increments every 16
clock cycles*/
setup_timer1(T1_EXTERNAL_RTC|T2_DIV_BY_16);

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

## Timer0

These options lets the user configure and use timer0. It is available on all devices and is always enabled. The clock/counter is 8-bit on PIC16and 8 or 16 bit on PIC18s. It counts up and also provides interrupt on overflow. The options available differ and are listed in the device header file.

**Relevant Functions:**
**setup_timer_0(mode) -** Sets the source, prescale etc for timer0

**set_timer0(value) or set_rtcc(value) -** Initializes the timer0 clock/counter. Value may be a 8-bit or 16-bit depending on the device.

v**alue=get_timer0 -** Returns the value of the timer0 clock/counter.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_TIMER0 or INT_RTCC - Interrupt fires when timer0 overflows.

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

**Relevant getenv() Parameters:**
TIMER0 - Returns 1 if the device has timer0

**Example Code:**
For PIC18F452:

```
  setup_timer_0(RTCC_INTERNAL|RTCC_DIV_@|RTCC_8_BIT);
                                  //sets the internal clock as source
                                  //and prescale 2.  At 20Mhz timer0
                                  //will increment every 0.4us in this
                                  //setup and overflows every 102.4us
  set_timer0(0);                   //this sets timer0 register to 0
  time-get_timer0();               this will read the timer0 register
  value
```

## Timer1

These options lets the user configure and use timer1. The clock/counter is 16-bit on PIC16s and PIC18s. It counts up and also provides interrupt on overflow. The options available differ and are listed in the device header file.

**Relevant Functions:**
**setup_timer_1(mode) -** Disables or sets the source and prescale for timer1.

s**et_timer1(value) -** Initializes the timer1 clock/counter.

v**alue=get_timer1 -** Returns the value of the timer1 clock/counter.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_TIMER1 - Interrupt fires when timer1 overflows

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

**Relevant getenv() Parameters:**
TIMER1 - Returns 1 if the device has timer1

**Example Code:**
For PIC18452:

```
setup_timer_1(T1_DISABLED);              //disables timer1
setup_timer_1(T1_INTERNAL|T1_DIV_BY_8); //sets the internal clock as
source
                                         //and prescale as 8.  At 20Mhz
timer1
                                         //will increment every 1.6us in
this
                                         //setup and overflows every
104.896ms
set_timer1(0);                           //this sets timer1 register to
0
time=get_timer1();                       //this will read the timer1
register value
```

## Timer2

These options lets the user configure and use timer2. The clock/counter is 8-bit on PIC16s and PIC18s. It counts up and also provides interrupt on overflow. The options available differ and are listed in the device header file.

**Relevant Functions:**
**setup_timer_2(mode,period,postscale)) -** Disables or sets the prescale, period and a postscale for timer2.

**set_timer2(value) -** Initializes the timer2 clock/counter.

v**alue=get_timer2 -** Returns the value of the timer2 clock/counter.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_TIMER2 - Interrupt fires when timer2 overflows

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

**Relevant getenv() Parameters:**
TIMER2 - Returns 1 if the device has timer2

**Example Code:**
For PIC18452:

```
setup_timer_2(T2_DISABLED);                    //disables timer2
setup_timer_2(T2_INTERNAL|T2_DIV_BY_4,0xc0,2); //sets the prescale as
4, period
                                               //as 0xc0 and postscales
```

```
as 2.
                                                    //At 20Mhz timer2 will
increment
                                                    // very .8us in this
setup
                                                    // and overflows every
154.4us
                                                  //and interrupt every
308.2us
set_timer2(0);                                     //this sets timer2
register to 0
time=get_timer2();                                //this will read timer2
register value
```

## Timer3

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

## Timer4

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

## Timer5

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

**Relevant Functions:**
**setup_timer_5(mode) -** Disables or sets the source and prescale for timer5.

**set_timer5(value) -** Initializes the timer5 clock/counter.

v**alue=get_timer5 -** Returns the value of the timer5 clock/counter

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_TIMER5 - Interrupt fires when timer5 overflows.

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

**Relevant getenv() Parameters:**
TIMER5 - Returns 1 if the device has timer5.

**Example Code:**
For PIC18F4431

```
  setup_timer_5(T5_DISABLED);              //disables timer5
  setup_timer_5(T5_INTERNAL|T5_DIV_BY_1); //sets the internal clock as
  source and
                                          //prescale as 1.  At 20Mhz
  timer5 will
                                          //increment every .2us in this
  setup
                                          //and overflows every 13.1072ms
  set_timer5(0);                           //this sets timer5 register to
  0
  time=get_timer5();                       //this will read the timer5
  register value
```

## TimerA

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

**Relevant Functions:**
**setup_timer_A(mode) -** Disable or sets the source and prescale for timerA.

**set_timerA(value) -** Initializes the timerA clock/counter.

v**alue=get_timerA() -** Returns the value of the timerA clock/counter

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_TIMERA - Interrupt fires timerA overflows

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

**Relevant getenv() Parameters:**
TIMERA - Returns 1 if the device has timerA

**Example Code:**

```
  setup_timer_A(TA_OFF);               //disable timerA
  setup_timer_A(TA_INTERNAL|TA_DIV_8); //sets the internal clock as
  source
                                      //and prescale as 8.  At 20Mhz
  timerA
```

```
                                        //will increment every 1.6us in
this
                                        //setup and overflows every 409.6us
set_timerA(0):                           //this sets timerA register to 0
time=get_timerA();                       //this will read the timerA
register value
```

## TimerB

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

**Relevant Functions:**
**setup_timer_B(mode) -** Disable or set the source and prescale for timerB.

s**et_timerB(value) -** Initializes the timerB clock/counter.

**value=get_timerB() -** Returns the value of the timerB clock/counter.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
INT_TIMERB - Interrupt fires when timerB overflows

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

**Relevant getenv() Parameters:**
TIMERB - Returns 1 if the device has timerB

**Example Code:**

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

## USB

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

**Relevant Functions:**

**usb_init() -** Initializes the USB hardware. Will then wait in an infinite loop for the USB peripheral to be connected to bus (but that doesn't mean it has been enumerated by the PC). Will enable and use the USB interrupt.

u**sb_init_cs() -** The same as usb_init(), but does not wait for the device to be connected to the bus. This is useful if your device is not bus powered and can operate without a USB connection.

u**sb_task() -** If you use connection sense, and the usb_init_cs() for initialization, then you must periodically call this function to keep an eye on the connection sense pin. When the PIC is connected to the BUS, this function will then perpare the USB peripheral. When the PIC is disconnected from the BUS, it will reset the USB stack and peripheral. Will enable and use the USB interrupt.

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

**usb_detach() -** Removes the PIC from the bus. Will be called automatically by usb_task() if connection is lost, but can be called manually by the user.

u**sb_attach() -** Attaches the PIC to the bus. Will be called automatically by usb_task() if connection is made, but can be called manually by the user.

u**sb_attached() -** If using connection sense pin (USB_CON_SENSE_PIN), returns TRUE if that pin is high. Else will always return TRUE.

u**sb_enumerated() -** Returns TRUE if the device has been enumerated by the PC. If the device has been enumerated by the PC, that means it is in normal operation mode and you can send/receive packets.

u**sb_put_packet(endpoint, data, len, tgl) -** Places the packet of data into the specified endpoint buffer. Returns TRUE if success, FALSE if the buffer is still full with the last packet.

u**sb_puts(endpoint, data, len,timeout) -** Sends the following data to the specified endpoint. usb_puts() differs from usb_put_packet() in that it will send multi packet messages if the data will not fit into one packet.

u**sb_kbhit(endpoint) -** Returns TRUE if the specified endpoint has data in it's receive buffer

u**sb_get_packet(endpoint, ptr, max) -** Reads up to max bytes from the specified endpoint buffer and saves it to the pointer ptr. Returns the number of bytes saved to ptr.

u**sb_gets(endpoint, ptr,max, timeout) -** Reads a message from the specified endpoint. The difference usb_get_packet() and usb_gets() is that usb_gets() will wait until a full message has received, which a message may contain more than one packet. Returns the number of bytes received.

**Relevant CDC Functions:**
A CDC USB device will emulate an RS-232 device, and will appear on your PC as a COM port. The follow functions provide you this virtual RS-232/serial interface.

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

**usb_cdc_kbhit() -** The same as kbhit(), returns TRUE if there is 1 or more character in the receive buffer.

u**sb_cdc_getc() -** The same as getc(), reads and returns a character from the receive buffer. If there is no data in the receive buffer it will wait indefinitely until there a character has been received.

u**sb_cdc_putc(c) -** The same as putc(), sends a character. It actually puts a character into the transmit buffer, and if the transmit buffer is full will wait indefinitely until there is space for the character.

u**sb_cdc_putc_fast(c) -** The same as usb_cdc_putc(), but will not wait indefinitely until there is space for the character in the transmit buffer. In that situation the character is lost.

u**sb_cdc_puts(*str) -** Sends a character string (null terminated) to the USB CDC port. Will return FALSE if the buffer is busy, TRUE if buffer is string was put into buffer for sending. Entire string must fit into endpoint, if string is longer than endpoint buffer then excess characters will be ignored.

u**sb_cdc_putready() -** Returns TRUE if there is space in the transmit buffer for another character.

**Relevant Preprocessor:**
None

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

**Relevant Include Files:**
**pic_usb.h -** Hardware layer driver for the PIC16C765 family PICmicro controllers with an internal USB peripheral.

p**ic18_usb.h -** Hardware layer driver for the PIC18F4550 family PICmicro controllers with an internal USB peripheral.

u**sbn960x.h -** Hardware layer driver for the National USBN9603/USBN9604 external USB peripheral. You can use this external peripheral to add USB to any microcontroller.

u**sb.h -** Common definitions and prototypes used by the USB driver.

u**sb.c -** The USB stack, which handles the USB interrupt and USB Setup Requests on Endpoint 0.

u**sb_cdc.h -** A driver that takes the previous include files to make a CDC USB device, which emulates an RS232 legacy device and shows up as a COM port in the MS Windows device manager.

**Relevant getenv() Parameters:**
USB - Returns TRUE if the device has an integrated internal USB peripheral.

**Example Code:**
Due to the complexity of USB example code will not fit here. But you can find the following examples installed with your CCS C Compiler:

- **ex_usb_hid.c -** A simple HID device
- **ex_usb_mouse.c -** A HID Mouse, when connected to the PC, the mouse cursor will go in circles.
- **ex_usb_kbmouse.c -** An example of how to create a USB device with multiple interfaces by creating a keyboard and mouse in one device.
- **ex_usb_kbmouse2.c -** An example of how to use multiple HID report IDs to transmit more than one type of HID packet, as demonstrated by a keyboard and mouse on one device.
- e**x_usb_scope.c -** A vendor-specific class using bulk transfers is demonstrated.
- e**x_usb_serial.c -** The CDC virtual RS232 library is demonstrated with this RS232 < - > USB example.
- e**x_usb_serial2.c -** Another CDC virtual RS232 library example, this time a port of the ex_intee.c example to use USB instead of RS232.

## Voltage Reference

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

**Relevant Functions:**
**setup_vref(mode | value) -** Enables and sets up the internal voltage reference value. Constants are defined in the device's .h file.

**Relevant Preprocessor:**
None

**Relevant Interrupts:**
None

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

**Relevant getenv() Parameters:**
VREF - Returns 1 if the device has VREF

**Example Code:**

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

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

## WDT or Watch Dog Timer

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

**Relevant Functions:**
**setup_wdt() -** Enables/disables the wdt or sets the prescalar.

r**estart_wdt() -** Restarts the wdt, if wdt is enables this must be periodically called to prevent a timeout reset.

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

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

**Relevant Preprocessor:**
#FUSES WDT/NOWDT - Enables/Disables WDT in PCB/PCM devices.

#FUSES WDT16 - Sets up the timeout/timein in PCH devices.

**Relevant Interrupts:**
None

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

**Relevant getenv() Parameters:**
None

**Example Code:**
For PIC16F877

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

For PIC18F452

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

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

## Stream I/O

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

**Ouptut:**
output:
stream << variable_or_constant_or_manipulator ;

_____
one or more repeats

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

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

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

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

**Manipulators:**
**hex** -Hex format numbers

**dec**- Decimal format numbers (default)

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

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

**boolalpha**- Output int1 as true and false

**noboolalpha** -Output int1 as 1 and 0 (default)

**fixed Floats**- in decimal format (default)

**scientific Floats**- use E notation

**iosdefault**- All manipulators to default settings

**endl** -Output CR/LF

**ends**- Outputs a null ('\000')

**Examples:**

```
cout << "Value is " << hex << data << endl;
```

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

**Input:**
stream >> variable_or_constant_or_manipulator ;

_____
one or more repeats

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

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

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

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

**Manipulators:**
**hex** -Hex format numbers

**dec**- Decimal format numbers (default)

**noecho**- Suppress echoing

**strspace**- Allow spaces to be input into strings

**nostrspace**- Spaces terminate string entry (default)

**iosdefault** -All manipulators to default settings

**Examples:**

```
cout << "Enter number: ";
cin >> value;
cout << "Enter title: ";
cin >> strspace >> title;
cin >> data[i].recordid >> data[i].xpos >> data[i].ypos >>
data[i].sample ;
```

```
string1 >> data;
lcdputc << "\fEnter count";
lcdputc << keypadgetc >> count;      // read from keypad, echo to lcd
                                     // This syntax only works with
                                     // user defined functions.
```

# PREPROCESSOR

## __address__

**Syntax:**
A predefined symbol _ _address_ _ may be used to indicate a type that must hold a program memory address.

**Examples:**

```
__address__ testa = 0x1000  //will allocate 16 bits for test a
and
                            //initialize to 0x1000
```

## _attribute_x

**Syntax:**
__attribute__x

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

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

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

On architectures where 'int' is 16bit (such as dsPIC or PIC24 microcontrollers), 'test' would take 4 bytes even though it is comprised of3 bytes. By applying the 'packed' attribute to this struct then it would take 3 bytes as originally intended:

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

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

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

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

**int8 array[256] __attribute__((aligned(0x1000)));**

This will tell the compiler to try to place 'array' at either 0x0, 0x1000, 0x2000, 0x3000, 0x4000, etc.

**Description:**
To alter some specifics as to how the compiler operates.

**Examples:**

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

## #asm, #endasm, #asm asis

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

**Elements:**
Code is a list of assembly language instructions.

**Description:**

| 12 Bit and 14 Bit | |
|---|---|
| **ADDWF f,d** | ANDWF f,d |
| **CLRF f** | CLRW |
| **COMF f,d** | DECF f,d |
| **DECFSZ f,d** | INCF f,d |
| **INCFSZ f,d** | IORWF f,d |
| **MOVF f,d** | MOVPHW |
| **MOVPLW** | MOVWF f |
| **NOP** | RLF f,d |
| **RRF f,d** | SUBWF f,d |
| **SWAPF f,d** | XORWF f,d |
| **BCF f,b** | BSF f,b |
| **BTFSC f,b** | BTFSS f,b |

| | |
|---|---|
| **ANDLW k** | CALL k |
| **CLRWDT** | GOTO k |
| **IORLW k** | MOVLW k |
| **RETLW k** | SLEEP |
| **XORLW** | OPTION |
| **TRIS k** | |
| | **14 Bit** |
| | ADDLW k |
| | SUBLW k |
| | RETFIE |
| | RETURN |

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

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

| PIC 18 | | | | | |
|---|---|---|---|---|---|
| **ADDWF** | f,d | ADDWFC | f,d | ANDWF | f,d |
| **CLRF** | f | COMF | f,d | CPFSEQ | f |
| **CPFSGT** | f | CPFSLT | f | DECF | f,d |
| **DECFSZ** | f,d | DCFSNZ | f,d | INCF | f,d |
| **INFSNZ** | f,d | IORWF | f,d | MOVF | f,d |
| **MOVFF** | fs,d | MOVWF | f | MULWF | f |
| **NEGF** | f | RLCF | f,d | RLNCF | f,d |
| **RRCF** | f,d | RRNCF | f,d | SETF | f |
| **SUBFWB** | f,d | SUBWF | f,d | SUBWFB | f,d |
| **SWAPF** | f,d | TSTFSZ | f | XORWF | f,d |
| **BCF** | f,b | BSF | f,b | BTFSC | f,b |
| **BTFSS** | f,b | BTG | f,d | BC | n |
| **BN** | n | BNC | n | BNN | n |
| **BNOV** | n | BNZ | n | BOV | n |
| **BRA** | n | BZ | n | CALL | n,s |
| **CLRWDT** | - | DAW | - | GOTO | n |
| **NOP** | - | NOP | - | POP | - |
| **PUSH** | - | RCALL | n | RESET | - |
| **RETFIE** | s | RETLW | k | RETURN | s |

| | | | | | |
|---|---|---|---|---|---|
| **SLEEP** | - | ADDLW | k | ANDLW | k |
| **IORLW** | k | LFSR | f,k | MOVLB | k |
| **MOVLW** | k | MULLW | k | RETLW | k |
| **SUBLW** | k | XORLW | k | TBLRD | * |
| **TBLRD** | *+ | TBLRD | *- | TBLRD | +* |
| **TBLWT** | * | TBLWT | *+ | TBLWT | *- |
| **TBLWT** | +* | | | | |

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

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

***[PCD]***

| PIC24 and dsPIC | | |
|---|---|---|
| **ADD** | Wa,Wb,Wd | Wd = Wa+Wb |
| **ADD** | f,W | W0 = f+Wd |
| **ADD** | lit10,Wd | Wd = lit10+Wd |
| **ADD** | Wa,lit5,Wd | Wd = lit5+Wa |
| **ADD** | f,F | f = f+Wd |
| **ADD** | acc | Acc = AccA+AccB |
| **ADD** | Wd,{lit4},acc | Acc = Acc+(Wa shifted slit4) |
| **ADD.B** | lit10,Wd | Wd = lit10+Wd (byte) |
| **ADD** | Wd,{lit4},acc | Acc = Acc+(Wa shifted slit4) |
| **ADD.B** | lit10,Wd | Wd = lit10+Wd (byte) |
| **ADD.B** | f,F | f = f+Wd (byte) |
| **ADD.B** | Wa,Wb,Wd | Wd = Wa+Wb (byte) |
| **ADD.B** | Wa,lit5,Wd | Wd = lit5+Wa (byte) |
| **ADD.B** | f,W | W0 = f+Wd (byte) |
| **ADDC** | f,W | Wd = f+Wa+C |
| **ADDC** | lit10,Wd | Wd = lit10+Wd+C |
| **ADDC** | Wa,lit5,Wd | Wd = lit5+Wa+C |
| **ADDC** | f,F | Wd = f+Wa+C |
| **ADDC** | Wa,Wb,Wd | Wd = Wa+Wb+C |
| **ADDC.B** | lit10,Wd | Wd = lit10+Wd+C (byte) |
| **ADDC.B** | Wa,Wb,Wd | Wd = Wa+Wb+C (byte) |
| **ADDC.B** | Wa,lit5,Wd | Wd = lit5+Wa+C (byte) |
| **ADDC.B** | f,W | Wd = f+Wa+C (byte) |
| **ADDC.B** | f,F | Wd = f+Wa+C (byte) |
| **AND** | Wa,Wb,Wd | Wd = Wa.&.Wb |

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

| | | |
|---|---|---|
| **BRA SB** | a | Branch if Accumulator B Saturate |
| **BRA Z** | a | Branch if Zero |
| **BREAK** | | ICD Break |
| **BSET** | Wd,B | Wa.bit = 1 |
| **BSET** | f,B | f.bit = 1 |
| **BSET.B** | Wd,B | Wa.bit = 1 (byte) |
| **BSW.C** | Wa,Wd | Wa.Wb = C |
| **BSW.Z** | Wa,Wd | Wa.Wb = Z |
| **BTG** | Wd,B | Wa.bit = ~Wa.bit |
| **BTG** | f,B | f.bit = ~f.bit |
| **BTG.B** | Wd,B | Wa.bit = ~Wa.bit (byte) |
| **BTSC** | f,B | Skip if f.bit = 0 |
| **BTSC** | Wd,B | Skip if Wa.bit4 = 0 |
| **BTSS** | f,B | Skip if f.bit = 1 |
| **BTSS** | Wd,B | Skip if Wa.bit = 1 |
| **BTST** | f,B | Z = f.bit |
| **BTST.C** | Wa,Wd | C = Wa.Wb |
| **BTST.C** | Wd,B | C = Wa.bit |
| **BTST.Z** | Wd,B | Z = Wa.bit |
| **BTST.Z** | Wa,Wd | Z = Wa.Wb |
| **BTSTS** | f,B | Z = f.bit; f.bit = 1 |
| **BTSTS.C** | Wd,B | C = Wa.bit; Wa.bit = 1 |
| **BTSTS.Z** | Wd,B | Z = Wa.bit; Wa.bit = 1 |
| **CALL** | a | Call subroutine |
| **CALL** | Wd | Call [Wa] |
| **CLR** | f,F | f = 0 |
| **CLR** | acc,da,dc,pi | Acc = 0; prefetch=0 |
| **CLR** | f,W | W0 = 0 |
| **CLR** | Wd | Wd = 0 |
| **CLR.B** | f,W | W0 = 0 (byte) |
| **CLR.B** | Wd | Wd = 0 (byte) |
| **CLR.B** | f,F | f = 0 (byte) |
| **CLRWDT** | | Clear WDT |
| **COM** | f,F | f = ~f |
| **COM** | f,W | W0 = ~f |
| **COM** | Wa,Wd | Wd = ~Wa |
| **COM.B** | f,W | W0 = ~f  (byte) |
| **COM.B** | Wa,Wd | Wd = ~Wa (byte) |
| **COM.B** | f,F | f = ~f  (byte) |
| **CP** | W,f | Status set for f - W0 |
| **CP** | Wa,Wd | Status set for Wb â€“ Wa |

| | | |
|---|---|---|
| **CP** | Wd,lit5 | Status set for Wa â€“ lit5 |
| **CP.B** | W,f | Status set for f - W0 (byte) |
| **CP.B** | Wa,Wd | Status set for Wb â€“ Wa (byte) |
| **CP.B** | Wd,lit5 | Status set for Wa â€“ lit5 (byte) |
| **CP0** | Wd | Status set for Wa â€“ 0 |
| **CP0** | W,f | Status set for f â€“ 0 |
| **CP0.B** | Wd | Status set for Wa â€“ 0 (byte) |
| **CP0.B** | W,f | Status set for f â€“ 0 (byte) |
| **CPB** | Wd,lit5 | Status set for Wa â€“ lit5 â€“ C |
| **CPB** | Wa,Wd | Status set for Wb â€“ Wa â€“ C |
| **CPB** | W,f | Status set for f â€“ W0 - C |
| **CPB.B** | Wa,Wd | Status set for Wb â€“ Wa â€“ C (byte) |
| **CPB.B** | Wd,lit5 | Status set for Wa â€“ lit5 â€“ C (byte) |
| **CPB.B** | W,f | Status set for f â€“ W0 - C (byte) |
| **CPSEQ** | Wa,Wd | Skip if Wa = Wb |
| **CPSEQ.B** | Wa,Wd | Skip if Wa = Wb (byte) |
| **CPSGT** | Wa,Wd | Skip if Wa > Wb |
| **CPSGT.B** | Wa,Wd | Skip if Wa > Wb (byte) |
| **CPSLT** | Wa,Wd | Skip if Wa < Wb |
| **CPSLT.B** | Wa,Wd | Skip if Wa < Wb (byte) |
| **CPSNE** | Wa,Wd | Skip if Wa != Wb |
| **CPSNE.B** | Wa,Wd | Skip if Wa != Wb (byte) |
| **DAW.B** | Wd | Wa = decimal adjust Wa |
| **DEC** | Wa,Wd | Wd = Wa â€“ 1 |
| **DEC** | f,W | W0 = f â€“ 1 |
| **DEC** | f,F | f = f â€“ 1 |
| **DEC.B** | f,F | f = f â€“ 1 (byte) |
| **DEC.B** | f,W | W0 = f â€“ 1 (byte) |
| **DEC.B** | Wa,Wd | Wd = Wa â€“ 1 (byte) |
| **DEC2** | Wa,Wd | Wd = Wa â€“ 2 |
| **DEC2** | f,W | W0 = f â€“ 2 |
| **DEC2** | f,F | f = f â€“ 2 |
| **DEC2.B** | Wa,Wd | Wd = Wa â€“ 2 (byte) |
| **DEC2.B** | f,W | W0 = f â€“ 2 (byte) |
| **DEC2.B** | f,F | f = f â€“ 2 (byte) |
| **DISI** | lit14 | Disable Interrupts lit14 cycles |
| **DIV.S** | Wa,Wd | Signed 16/16-bit integer divide |
| **DIV.SD** | Wa,Wd | Signed 16/16-bit integer divide (dword) |
| **DIV.U** | Wa,Wd | UnSigned 16/16-bit integer divide |
| **DIV.UD** | Wa,Wd | UnSigned 16/16-bit integer divide (dword) |
| **DIVF** | Wa,Wd | Signed 16/16-bit fractional divide |

| | | |
|---|---|---|
| **DO** | lit14,a | Do block lit14 times |
| **DO** | Wd,a | Do block Wa times |
| **ED** | Wd*Wd,acc,da,db | Euclidean Distance (No Accumulate) |
| **EDAC** | Wd*Wd,acc,da,db | Euclidean Distance |
| **EXCH** | Wa,Wd | Swap Wa and Wb |
| **FBCL** | Wa,Wd | Find bit change from left (Msb) side |
| **FEX** | | ICD Execute |
| **FF1L** | Wa,Wd | Find first one from left (Msb) side |
| **FF1R** | Wa,Wd | Find first one from right (Lsb) side |
| **GOTO** | a | GoTo |
| **GOTO** | Wd | GoTo [Wa] |
| **INC** | f,W | W0 = f + 1 |
| **INC** | Wa,Wd | Wd = Wa + 1 |
| **INC** | f,F | f = f + 1 |
| **INC.B** | Wa,Wd | Wd = Wa + 1 (byte) |
| **INC.B** | f,F | f = f + 1 (byte) |
| **INC.B** | f,W | W0 = f + 1 (byte) |
| **INC2** | f,W | W0 = f + 2 |
| **INC2** | Wa,Wd | Wd = Wa + 2 |
| **INC2** | f,F | f = f + 2 |
| **INC2.B** | f,W | W0 = f + 2 (byte) |
| **INC2.B** | f,F | f = f + 2 (byte) |
| **INC2.B** | Wa,Wd | Wd = Wa + 2 (byte) |
| **IOR** | lit10,Wd | Wd = lit10 \| Wd |
| **IOR** | f,F | f = f \| Wa |
| **IOR** | f,W | W0 = f \| Wa |
| **IOR** | Wa,lit5,Wd | Wd = Wa.\|.lit5 |
| **IOR** | Wa,Wb,Wd | Wd = Wa.\|.Wb |
| **IOR.B** | Wa,Wb,Wd | Wd = Wa.\|.Wb (byte) |
| **IOR.B** | f,W | W0 = f \| Wa (byte) |
| **IOR.B** | lit10,Wd | Wd = lit10 \| Wd (byte) |
| **IOR.B** | Wa,lit5,Wd | Wd = Wa.\|.lit5 (byte) |
| **IOR.B** | f,F | f = f \| Wa (byte) |
| **LAC** | Wd,{lit4},acc | Acc = Wa shifted slit4 |
| **LNK** | lit14 | Allocate Stack Frame |
| **LSR** | f,W | W0 = f >> 1 |
| **LSR** | Wa,lit4,Wd | Wd = Wa >> lit4 |
| **LSR** | Wa,Wd | Wd = Wa >> 1 |
| **LSR** | f,F | f = f >> 1 |
| **LSR** | Wa,Wb,Wd | Wd = Wb >> Wa |
| **LSR.B** | f,W | W0 = f >> 1 (byte) |

| | | |
|---|---|---|
| **LSR.B** | f,F | f = f >> 1 (byte) |
| **LSR.B** | Wa,Wd | Wd = Wa >> 1 (byte) |
| **MAC** | Wd*Wd,acc,da,dc | Acc = Acc + Wa * Wa; {prefetch} |
| **MAC** | Wd*Wc,acc,da,dc,pi | Acc = Acc + Wa * Wb; {[W13] = Acc}; {prefetch} |
| **MOV** | W,f | f = Wa |
| **MOV** | f,W | W0 = f |
| **MOV** | f,F | f = f |
| **MOV** | Wd,? | F = Wa |
| **MOV** | Wa+lit,Wd | Wd = [Wa +Slit10] |
| **MOV** | ?,Wd | Wd = f |
| **MOV** | lit16,Wd | Wd = lit16 |
| **MOV** | Wa,Wd | Wd = Wa |
| **MOV** | Wa,Wd+lit | [Wd + Slit10] = Wa |
| **MOV.B** | lit8,Wd | Wd = lit8 (byte) |
| **MOV.B** | W,f | f = Wa (byte) |
| **MOV.B** | f,W | W0 = f (byte) |
| **MOV.B** | f,F | f = f (byte) |
| **MOV.B** | Wa+lit,Wd | Wd = [Wa +Slit10] (byte) |
| **MOV.B** | Wa,Wd+lit | [Wd + Slit10] = Wa (byte) |
| **MOV.B** | Wa,Wd | Wd = Wa (byte) |
| **MOV.D** | Wa,Wd | Wd:Wd+1 = Wa:Wa+1 |
| **MOV.D** | Wa,Wd | Wd:Wd+1 = Wa:Wa+1 |
| **MOVSAC** | acc,da,dc,pi | Move ? to ? and ? To ? |
| **MPY** | Wd*Wc,acc,da,dc | Acc = Wa*Wb |
| **MPY** | Wd*Wd,acc,da,dc | Square to Acc |
| **MPY.N** | Wd*Wc,acc,da,dc | Acc = -(Wa*Wb) |
| **MSC** | Wd*Wc,acc,da,dc,pi | Acc = Acc â€“ Wa*Wb |
| **MUL** | W,f | W3:W2 = f * Wa |
| **MUL.B** | W,f | W3:W2 = f * Wa (byte) |
| **MUL.SS** | Wa,Wd | {Wd+1,Wd}= sign(Wa) * sign(Wb) |
| **MUL.SU** | Wa,Wd | {Wd+1,Wd} = sign(Wa) * unsign(Wb) |
| **MUL.SU** | Wa,lit5,Wd | {Wd+1,Wd}= sign(Wa) * unsign(lit5) |
| **MUL.US** | Wa,Wd | {Wd+1,Wd} = unsign(Wa) * sign(Wb) |
| **MUL.UU** | Wa,Wd | {Wd+1,Wd} = unsign(Wa) * unsign(Wb) |
| **MUL.UU** | Wa,lit5,Wd | {Wd+1,Wd} = unsign(Wa) * unsign(lit5) |
| **NEG** | f,F | f = - f |
| **PUSH** | Wd | Push Wa to TOS |
| **PUSH.D** | Wd | PUSH double Wa:Wa + 1 to TOS |
| **PUSH.S** | | PUSH shadow registers |
| **PWRSAV** | lit1 | Enter Power-saving mode lit1 |
| **RCALL** | a | Call (relative) |

| | | |
|---|---|---|
| **RCALL** | Wd | Call Wa |
| **REPEAT** | lit14 | Repeat next instruction (lit14 + 1) times |
| **REPEAT** | Wd | Repeat next instruction (Wa + 1) times |
| **RESET** | | Reset |
| **RETFIE** | | Return from interrupt enable |
| **RETLW** | lit10,Wd | Return; Wa = lit10 |
| **RETLW.B** | lit10,Wd | Return; Wa = lit10 (byte) |
| **RETURN** | | Return |
| **RLC** | Wa,Wd | Wd = rotate left through Carry Wa |
| **RLC** | f,F | f = rotate left through Carry f |
| **RLC** | f,W | W0 = rotate left through Carry f |
| **RLC.B** | f,F | f = rotate left through Carry f (byte) |
| **RLC.B** | f,W | W0 = rotate left through Carry f (byte) |
| **RLC.B** | Wa,Wd | Wd = rotate left through Carry Wa (byte) |
| **RLNC** | Wa,Wd | Wd = rotate left (no Carry) Wa |
| **RLNC** | f,F | f = rotate left (no Carry) f |
| **RLNC** | f,W | W0 = rotate left (no Carry) f |
| **RLNC.B** | f,W | W0 = rotate left (no Carry) f (byte) |
| **RLNC.B** | Wa,Wd | Wd = rotate left (no Carry) Wa (byte) |
| **RLNC.B** | f,F | f = rotate left (no Carry) f (byte) |
| **RRC** | f,F | f = rotate right through Carry f |
| **RRC** | Wa,Wd | Wd = rotate right through Carry Wa |
| **RRC** | f,W | W0 = rotate right through Carry f |
| **RRC.B** | f,W | W0 = rotate right through Carry f (byte) |
| **RRC.B** | f,F | f = rotate right through Carry f (byte) |
| **RRC.B** | Wa,Wd | Wd = rotate right through Carry Wa (byte) |
| **RRNC** | f,F | f = rotate right (no Carry) f |
| **RRNC** | f,W | W0 = rotate right (no Carry) f |
| **RRNC** | Wa,Wd | Wd = rotate right (no Carry) Wa |
| **RRNC.B** | f,F | f = rotate right (no Carry) f (byte) |
| **RRNC.B** | Wa,Wd | Wd = rotate right (no Carry) Wa (byte) |
| **RRNC.B** | f,W | W0 = rotate right (no Carry) f (byte) |
| **SAC** | acc,{lit4},Wd | Wd = Acc slit 4 |
| **SAC.R** | acc,{lit4},Wd | Wd = Acc slit 4 with rounding |
| **SE** | Wa,Wd | Wd = sign-extended Wa |
| **SETM** | Wd | Wd = 0xFFFF |
| **SETM** | f,F | W0 = 0xFFFF |
| **SETM.B** | Wd | Wd = 0xFFFF (byte) |
| **SETM.B** | f,W | W0 = 0xFFFF (byte) |
| **SETM.B** | f,F | W0 = 0xFFFF (byte) |
| **SFTAC** | acc,Wd | Arithmetic shift Acc by (Wa) |

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

| | | |
|---|---|---|
| **SUBR** | Wa,Wb,Wd | Wd = Wa â€“ Wb |
| **SUBR** | f,W | W0 = W0 â€“ f |
| **SUBR.B** | Wa,Wb,Wd | Wd = Wa â€“ Wb (byte) |
| **SUBR.B** | f,F | f = W0 â€“ f (byte) |
| **SUBR.B** | Wa,lit5,Wd | Wd = lit5 â€“ Wb (byte) |
| **SUBR.B** | f,W | W0 = W0 â€“ f (byte) |
| **SWAP** | Wd | Wa = byte or nibble swap Wa |
| **SWAP.B** | Wd | Wa = byte or nibble swap Wa (byte) |
| **TBLRDH** | Wa,Wd | Wd = ROM[Wa] for odd ROM |
| **TBLRDH.B** | Wa,Wd | Wd = ROM[Wa] for odd ROM (byte) |
| **TBLRDL** | Wa,Wd | Wd = ROM[Wa] for even ROM |
| **TBLRDL.B** | Wa,Wd | Wd = ROM[Wa] for even ROM (byte) |
| **TBLWTH** | Wa,Wd | ROM[Wa] = Wd for odd ROM |
| **TBLWTH.B** | Wa,Wd | ROM[Wa] = Wd for odd ROM (byte) |
| **TBLWTL** | Wa,Wd | ROM[Wa] = Wd for even ROM |
| **TBLWTL.B** | Wa,Wd | ROM[Wa] = Wd for even ROM (byte) |
| **ULNK** | | Deallocate Stack Frame |
| **URUN** | | ICD Run |
| **XOR** | Wa,Wb,Wd | Wd = Wa ^ Wb |
| **XOR** | f,F | f = f ^ W0 |
| **XOR** | f,W | W0 = f ^ W0 |
| **XOR** | Wa,lit5,Wd | Wd = Wa ^ lit5 |
| **XOR** | lit10,Wd | Wd = Wd ^ lit10 |
| **XOR.B** | lit10,Wd | Wd = Wd ^ lit10 (byte) |
| **XOR.B** | f,W | W0 = f ^ W0 (byte) |
| **XOR.B** | Wa,lit5,Wd | Wd = Wa ^ lit5 (byte) |
| **XOR.B** | Wa,Wb,Wd | Wd = Wa ^ Wb (byte) |
| **XOR.B** | f,F | f = f ^ W0 (byte) |
| **ZE** | Wa,Wd | Wd = Wa & FF |

**Example Files:**
**FFT.c**

**Examples:**

```
int find_parity(int data){

    int count;
    #asm
    MOV #0x08, W0
    MOV W0, count
    CLR W0
```

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

## #bank_dma

**Syntax:**
#bank_dma

**Elements:**
None

**Description:**
Informs the compiler to assign the data for the next variable, array or structure into DMA bank.

**Examples:**

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

## #bankx

**Syntax:**
#bankx

None

**Description:**
Informs the compiler to assign the data for the next variable, array or structure into BankX.

**Examples:**

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

## #banky

**Syntax:**
#banky

None

**Description:**
Informs the compiler to assign the data for the next variable, array or structure into BankY.

**Examples:**

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

## #bit

**Syntax:**
#BIT ***id*** = ***x***.***y***

**Elements:**
***id*** is a valid C identifier,
***x*** is a constant or a C variable,
***y*** is a constant 0-7 (for 8-bit PICs)
*[PCD]* ***y*** is a constant 0-15

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

**Example Files:**
ex_glint.c

**Examples:**
```
#bit T0IF = 0x b.2
...
T1IF = 0; // Clear Timer 0 interrupt flag

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

*[PCD]*
```
#bit T1IF = 0x84.3
...
T1IF = 0; // Clear Timer 0 interrupt flag

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

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

## __buildcount__

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

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

## #build

**Syntax:**
#BUILD(***segment*** = ***address***)
#BUILD(***segment*** = ***address***, ***segment*** = ***address***)
#BUILD(***segment*** = ***start***:***end***)
#BUILD(***segment*** = ***start***: ***end***, ***segment*** = ***start***: ***end***)
#BUILD(***nosleep***)
*[PCD]* #BUILD(***segment = size***) : For STACK use only

*[PCD]* #BUILD(**ALT_INTERRUPT**)
*[PCD]* #BUILD(AUX_MEMORY)

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

*[PCD]* ***segment -*** is one of the following memory segments which may be assigned a location: RESET, INTERRUPT, or STACK.

***address -*** is a ROM location memory address. ***Start*** and ***end*** are used to specify a range in memory to be used.

***start -*** is the first ROM location and ***end*** is the last ROM location to be used.

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

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

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

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

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

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

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

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

**Description:**
PIC18XXX devices with external ROM or PIC18XXX devices with no internal ROM can direct the compiler to utilize the ROM. When linking multiple compilation units, this directive must appear exactly the same in each compilation unit.

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

**Example Files:**
ex_glint.c

**Examples:**

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

*[PCD]*

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

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

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

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

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

# #byte

**Syntax:**
#byte ***id*** = ***x***

**Elements:**
***id*** is a valid C identifier,
***x*** is a C variable or a constant

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

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

**Example Files:**
ex_glint.c

**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;
```

*[PCD]*

```
#byte  status _register = 0x42
#byte  b_port = 0x02C8

struct  {
  short int r_w;
  short int c_d;
    int data    : 6 ; } E _port;
#byte a_port = 0x2DA
...
a_port.c_d = 1;
```

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

## #case

**Syntax:**
#case

**Elements:**
*None*

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

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

**Example Files:**
ex_cust.c

**Examples:**
```
#case

int STATUS;

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

## __date__

**Syntax:**
__date__

**Elements:**
None

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

**Example Files:**
ex_glint.c

**Examples:**
```
printf("Software was compiled on ");
printf(__DATE__)
```

## #define

**Syntax:**
#define ***id*** text
or
#define ***id***(***x***,***y***...) text

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

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

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

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

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

Within the define text two special operators are supported:
#x is the stringize operator resulting in "x"
x##y is the concatination operator resulting in xy

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

**Example Files:**
ex_stwt.c, ex_macro.c

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


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

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

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

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

**See Also:**
#UNDEF, #IFDEF, #IFNDEF

## #definedinc

**Syntax:**
value = definedinc( ***variable*** );

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

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

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

**Availability:**
All Device

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

## #device

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

**Elements:**
***Chip Options:***
***chip*** is the name of a specific processor (like: PIC16C74 or dsPIC33FJ64GP306), To get a current list of supported devices: START | RUN | CCSC +Q
***Options*** are qualifiers to the standard operation of the device. Valid options are:

| | |
|---|---|
| *=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 |
| *[PCD]* **ADC=SIGNED** | Result returned from read_adc() is signed.(Default is unsigned) |
| *[PCD]* **ADC=UNSIGNED** | Return result from read_adc() is unsigned.(default is UNSIGNED) |
| **ICD=TRUE** | Generates code compatible with Microchips ICD debugging hardware. |
| **ICD=n** | For chips with multiple ICSP ports specify the port number being used. The default is 1. |
| **WRITE_EEPROM=ASYNC** | Prevents WRITE_EEPROM from hanging while writing is taking place. When used, do not write to EEPROM from both ISR and outside ISR. |
| **WRITE_EEPROM = NOINT** | Allows interrupts to occur while the write_eeprom() operations is polling the done bit to check if the write operations has completed. Can be used as long as no EEPROM operations are performed during an ISR. |
| **HIGH_INTS=TRUE** | Use this option for high/low priority interrupts on the PIC® 18. |
| **%f=.** | No 0 before a decimal pint on %f numbers less than 1. |
| **OVERLOAD=KEYWORD** | Overloading of functions is now supported. Requires the |

| | |
|---|---|
| | use of the keyword for overloading. |
| **OVERLOAD=AUTO** | Default mode for overloading. |
| **PASS_STRINGS=IN_RAM** | A new way to pass constant strings to a function by first copying the string to RAM and then passing a pointer to RAM to the function. |
| **CONST=READ_ONLY** | Uses the ANSI keyword CONST definition, making CONST variables read only, rather than located in program memory. |
| **CONST=ROM** | Uses the CCS compiler traditional keyword CONST definition, making CONST variables located in program memory. |
| **NESTED_INTERRUPTS=TRUE** | Enables interrupt nesting for PIC24, dsPIC30, and dsPIC33 devices. Allows higher priority interrupts to interrupt lower priority interrupts. |
| **NORETFIE** | ISR functions (preceded by a #int_xxx) will use a RETURN opcode instead of the RETFIE opcode. This is not a commonly used option; used rarely in cases where the user is writing their own ISR handler. |
| **NO_DIGITAL_INIT** | Normally the compiler sets all I/O pins to digital and turns off the comparator. This option prevents that action. |
| **VECTOR_INTS** | For devices with both single and multiple vector interrupts. This selects multiple vectors. |
| *[PCD]* **DUAL_PARTITION** | For devices with Dual Partition Flash Modes, this enables Dual Partition Flash mode by setting the FBOOT configuration register to the appropriate value. It cuts the available program memory in half, and moves the configuration register addresses to the Dual Partition locations. |
| *[PCD]* **DUAL_PARTITION_PROTECTED** | For devices with Dual Partition Flash Modes this enabled Protected Dual Partition Flash mode, Partition 1 is write-protected when inactive, by setting the FBOOT configuration register to the appropriate value. It cuts the available program memory in half and moves the configuration register addresses to the Dual Partition locations. |
| *[PCD]* **PARTITION_SEQUENCE=x** | A value from 0 to 4095 to set the FBTSEQ configuration register. Only used when either DUAL_PARTITION or DUAL_PARTITION_PROTECTED is used. The value is used to determine which partition is active on power-up. The Partition with the lowest value will be the active partition. If the value is the same for both partitions, then Partition 1 will be the active partition on power-up. |

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

**Compilation mode selection:**

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

| | |
|---|---|
| **CCS4** | **This is the default compilation mode. The pointer size in this mode for PCM and PCH is set to *=16 if the part has RAM over 0FF.** |
| **ANSI** | Default data type is SIGNED all other modes default is UNSIGNED. Compilation is case sensitive, all other modes are case insensitive. Pointer size is set to *=16 if the part has RAM over 0FF. |
| **CCS2 CCS3** | var16 = NegConst8 is compiled as: var16 = NegConst8 & 0xff (no sign extension)<br>Pointer size is set to *=8 for PCM and PCH and *=5 for PCB . The overload keyword is required. |
| **CCS2 only** | The default #DEVICE ADC is set to the resolution of the part, all other modes default to 8.<br>onebit = eightbits is compiled as onebit = (eightbits != 0)<br>All other modes compile as: onebit = (eightbits & 1) |

**Description:**
To alter some specifics as to how the compiler operates

**Example Files:**
ex_mxram.c , ex_icd.c , 16c74.h

**Examples:**
***Chip Options:***

```
#device PIC16C74
#device PIC16C67 *=16
#device *=16  ICD=TRUE
#device PIC16F877 *=16 ADC=10
#device %f=.
printf("%f",.5); //will print .5, without the directive it will print 0.5
[PCD] #device DSPIC33FJ64GP306
[PCD] #device PIC24FJ64GA002 ICD=TRUE
[PCD] #device ADC=10
[PCD] #device ICD=TRUE ADC=10

[PCD] Float Options-
[PCD] #device %f=.
[PCD] printf("%f",.5); //will print .5, without the directive it will print 0.5
```

**Compilation mode selection:**
#device CCS2  // This will set the ADC to the resolution of the part

**See Also:**
read_adc()

## __device__

**Syntax:**
__device__

**Elements:**
None

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

**Examples:**
```
#if__device__==71
SETUP_ADC_PORTS(All_DIGITAL);
#endif
```

**See Also:**
#DEVICE

## #if #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.

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

Note: you may NOT use C variables in the #IF. Only preprocessor identifiers created via #define can be used.
The preprocessor expression DEFINED(id) may be used to return 1 if the id is defined and 0 if it is not.
== and != operators now accept a constant string as both operands. This allows for compile time comparisons and can be used with GETENV() when it returns a string result.

**Example Files:**
ex_extee.c

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

**See Also:**
#IFDEF, #IFNDEF, getenv()

## #error

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

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

**Description:**
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.

**Example Files:**
ex_psp.

**Examples:**

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

**See Also:**
#WARNING

## #export (options)

**Syntax:**
#export(options)

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

**Output Formats:**
***C -*** Indicates the file format is C source code. In this case the object is not exported but rather a definition that allows another C program in the same memory space to call the exported functions. It may be used by a bootloader that needs the loaded application to call bootloader functions.

***RELOCATABLE -*** CCS relocatable object file format. Must be imported or linked before loading into a PIC. This is the default format when the #EXPORT is used.

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

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

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

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

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

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

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

**Description:**
This directive will tell the compiler to either generate a relocatable object file or a stand-alone HEX binary. A relocatable object file must be linked into your application, while a stand-alone HEX binary can be programmed directly into the device. The command line compiler and the PCW IDE Project Manager can also be used to compile/link/build modules and/or projects. Multiple #EXPORT directives may be used to generate multiple hex files. This may be used for 18F8722 like devices with external memory.

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

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

## __file__

**Syntax:**
__file__

**Elements:**
None

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

**Example Files:**
assert.h

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

**See Also:**
__line__

## __filename__

**Syntax:**
__filename__

**Elements:**
None

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

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

**See Also:**
__line__

## #fill_rom

**Syntax:**
#fill_rom ***value***

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

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

**Example Files:**
ex_glint.c

**Examples:**
```
#fill_rom 0x36
```

**See Also:**
#ROM

## #fuses

**Syntax:**
#fuses ***options***

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

- LP, XT, HS, RC
- WDT, NOWDT
- PROTECT, NOPROTECT
- PUT, NOPUT (Power Up Timer)
- BROWNOUT, NOBROWNOUT

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

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

Some processors allow different levels for certain fuses. To access these levels, assign a value to the fuse. For example, on the 18F452, the fuse PROTECT=6 would place the value 6 into CONFIG5L, protecting code blocks 0 and 3.

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

To eliminate all fuses in the output files use: **#FUSES none**

To manually set the fuses in the output files use: **#FUSES 1 = 0xC200 // sets config word 1 to 0xC200**

**Example Files:**
ex_sqw.c

**Examples:**
```
  #fuses  HS,NOWDT
```

## #hexcomment

**Syntax:**
#HEXCOMMENT text comment for the top of the hex file
#HEXCOMMENT\ text comment for the end of the hex file

**Elements:**
None

**Description:**
Puts a comment in the hex file.
Some programmers (MPLAB in particular) do not like comments at the top of the hex file.

**Examples:**
```
  #hexcommentVersion3.1 - requires 20Mhz crystal
```

## #id

**Syntax:**
#ID ***number 16***
*[PCD]* #ID ***number 32***
#ID ***number***, ***number***, ***number***, ***number***
#ID ***"filename"***
#ID ***CHECKSUM***

**Elements:**
*Number 16* is a 16 bit number, *number* is a 4 bit number. [PCD] Number 3 *2* is a 32 bit number, *number* is a 8 bit number. Filename is any valid PC filename and *checksum* is a keyword.

**Description:**
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 ([PCD] 32)-bit number and put one nibble ([PCD] byte) in each of the four ID words ([PCD] bytes) in the traditional manner. The second syntax specifies the exact value to be used in each of the four ID words ([PCD] bytes).

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

**Example Files:**
ex_cust.c

**Examples:**
```
  #id  0x1234
  #id  "serial.num"
  #id  CHECKSUM
```

([PCD]
```
  #id 0x12345678
  #id 0x12, 0x34, 0x45, 0x67
  #id  "serial.num"
  #id  CHECKSUM
```

## #ifdef #ifndef #else #endif

**Syntax:**
#ifdef ***id***
  ***code***
#elif
  ***code***
#else //optional
  ***code***
#endif

#ifndef ***id***
  ***code***
#elif
  ***code***

#else //optional
***code***
#endif

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

**Description:**
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.

**Example Files:**
ex_sqw.c

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

**See Also:**
#IF

## #ignore_warnings

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

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

**Description:**
This function will suppress warning messages from the compiler. ALL indicates no warning will be generated. NONE indicates all warnings will be generated. If numbers are listed then those warnings are suppressed

**Example Files:**
ex_glint.c

**Examples:**

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

**See Also:**
Warning messages

## #import(options)

**Syntax:**
#import(options)

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

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

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

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

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

**HEX** - Imported data is straight hex data.

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

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

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

**Description:**
This directive will tell the compiler to include (link) a relocatable object with this unit during compilation. Normally all global symbols from the specified file will be linked, but the EXCEPT and ONLY options can prevent certain symbols from being linked.
The command line compiler and the PCW IDE Project Manager can also be used to compile/link/build modules and/or projects.

**Example Files:**
ex_glint.c

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

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

## #include

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

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

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

**Example Files:**
ex_sqw.c

**Examples:**
```
#include  <16C54.H>

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

## #inline

**Syntax:**
#inline

**Elements:**
None

**Description:**
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.

**Example Files:**
ex_cust.c

**Examples:**

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

**See Also:**
#SEPARATE

## #int_xxxx

**Syntax:**
**PCB, PCM, PCH**

| | |
|---|---|
| #INT_AD | **Analog to digital conversion complete** |
| #INT_ADOF | Analog to digital conversion timeout |
| #INT_BUSCOL | Bus collision |
| #INT_BUSCOL2 | Bus collision 2 detected |
| #INT_BUTTON | Pushbutton |
| #INT_CANERR | An error has occurred in the CAN module |
| #INT_CANIRX | An invalid message has occurred on the CAN bus |
| #INT_CANRX0 | CAN Receive buffer 0 has received a new message |

| | |
|---|---|
| #INT_CANRX1 | CAN Receive buffer 1 has received a new message |
| #INT_CANTX0 | CAN Transmit buffer 0 has completed transmission |
| #INT_CANTX1 | CAN Transmit buffer 0 has completed transmission |
| #INT_CANTX2 | CAN Transmit buffer 0 has completed transmission |
| #INT_CANWAKE | Bus Activity wake-up has occurred on the CAN bus |
| #INT_CCP1 | Capture or Compare on unit 1 |
| #INT_CCP2 | Capture or Compare on unit 2 |
| #INT_CCP3 | Capture or Compare on unit 3 |
| #INT_CCP4 | Capture or Compare on unit 4 |
| #INT_CCP5 | Capture or Compare on unit 5 |
| #INT_COMP | Comparator detect |
| #INT_COMP0 | Comparator 0 detect |
| #INT_COMP1 | Comparator 1 detect |
| #INT_COMP2 | Comparator 2 detect |
| #INT_CR | Cryptographic activity complete |
| #INT_EEPROM | Write complete |
| #INT_ETH | Ethernet module interrupt |
| #INT_EXT | External interrupt |
| #INT_EXT1 | External interrupt #1 |
| #INT_EXT2 | External interrupt #2 |
| #INT_EXT3 | External interrupt #3 |
| #INT_I2C | I2C interrupt (only on 14000) |
| #INT_IC1 | Input Capture #1 |
| #INT_IC2QEI | Input Capture 2 / QEI Interrupt |
| #IC3DR | Input Capture 3 / Direction Change Interrupt |
| #INT_LCD | LCD activity |
| #INT_LOWVOLT | Low voltage detected |
| #INT_LVD | Low voltage detected |
| #INT_OSC_FAIL | System oscillator failed |
| #INT_OSCF | System oscillator failed |

| | |
|---|---|
| #INT_PMP | Parallel Master Port interrupt |
| #INT_PSP | Parallel Slave Port data in |
| #INT_PWMTB | PWM Time Base |
| #INT_RA | Port A any change on A0_A5 |
| #INT_RB | Port B any change on B4-B7 |
| #INT_RC | Port C any change on C4-C7 |
| #INT_RDA | RS232 receive data available |
| #INT_RDA0 | RS232 receive data available in buffer 0 |
| #INT_RDA1 | RS232 receive data available in buffer 1 |
| #INT_RDA2 | RS232 receive data available in buffer 2 |
| #INT_RTCC | Timer 0 (RTCC) overflow |
| #INT_SPP | Streaming Parallel Port Read/Write |
| #INT_SSP | SPI or I2C activity |
| #INT_SSP2 | SPI or I2C activity for Port 2 |
| #INT_TBE | RS232 transmit buffer empty |
| #INT_TBE0 | RS232 transmit buffer 0 empty |
| #INT_TBE1 | RS232 transmit buffer 1 empty |
| #INT_TBE2 | RS232 transmit buffer 2 empty |
| #INT_TIMER0 | Timer 0 (RTCC) overflow |
| #INT_TIMER1 | Timer 1 overflow |
| #INT_TIMER2 | Timer 2 overflow |
| #INT_TIMER3 | Timer 3 overflow |
| #INT_TIMER4 | Timer 4 overflow |
| #INT_TIMER5 | Timer 5 overflow |
| #INT_ULPWU | Ultra-low power wake up interrupt |
| #INT_USB | Universal Serial Bus activity |

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

### *[PCD]* PCD (PIC24/dsPIC devices)

| | |
|---|---|
| **#INT_AC1** | **Analog comparator 1 output change** |
| **#INT_AC2** | Analog comparator 2 output change |

| | |
|---|---|
| **#INT_AC3** | Analog comparator 3 output change |
| **#INT_AC4** | Analog comparator 4 output change |
| **#INT_ADC1** | ADC1 conversion complete |
| **#INT_ADC2** | Analog to digital conversion complete |
| **#INT_ADCP0** | ADC pair 0 conversion complete |
| **#INT_ADCP1** | ADC pair 1 conversion complete |
| **#INT_ADCP2** | ADC pair 2 conversion complete |
| **#INT_ADCP3** | ADC pair 3 conversion complete |
| **#INT_ADCP4** | ADC pair 4 conversion complete |
| **#INT_ADCP5** | ADC pair 5 conversion complete |
| **#INT_ADDRERR** | Address error trap |
| **#INT_C1RX** | ECAN1 Receive Data Ready |
| **#INT_C1TX** | ECAN1 Transmit Data Request |
| **#INT_C2RX** | ECAN2 Receive Data Ready |
| **#INT_C2TX** | ECAN2 Transmit Data Request |
| **#INT_CAN1** | CAN 1 Combined Interrupt Request |
| **#INT_CAN2** | CAN 2 Combined Interrupt Request |
| **#INT_CNI** | Input change notification interrupt |
| **#INT_COMP** | Comparator event |
| **#INT_CRC** | Cyclic redundancy check generator |
| **#INT_DCI** | DCI transfer done |
| **#INT_DCIE** | DCE error |
| **#INT_DMA0** | DMA channel 0 transfer complete |
| **#INT_DMA1** | DMA channel 1 transfer complete |
| **#INT_DMA2** | DMA channel 2 transfer complete |
| **#INT_DMA3** | DMA channel 3 transfer complete |
| **#INT_DMA4** | DMA channel 4 transfer complete |
| **#INT_DMA5** | DMA channel 5 transfer complete |
| **#INT_DMA6** | DMA channel 6 transfer complete |
| **#INT_DMA7** | DMA channel 7 transfer complete |
| **#INT_DMAERR** | DMAC error trap |
| **#INT_EEPROM** | Write complete |
| **#INT_EX1** | External Interrupt 1 |
| **#INT_EX4** | External Interrupt 4 |
| **#INT_EXT0** | External Interrupt 0 |
| **#INT_EXT1** | External interrupt #1 |
| **#INT_EXT2** | External interrupt #2 |

| | |
|---|---|
| **#INT_EXT3** | External interrupt #3 |
| **#INT_EXT4** | External interrupt #4 |
| **#INT_FAULTA** | PWM Fault A |
| **#INT_FAULTA2** | PWM Fault A 2 |
| **#INT_FAULTB** | PWM Fault B |
| **#INT_IC1** | Input Capture #1 |
| **#INT_IC2** | Input Capture #2 |
| **#INT_IC3** | Input Capture #3 |
| **#INT_IC4** | Input Capture #4 |
| **#INT_IC5** | Input Capture #5 |
| **#INT_IC6** | Input Capture #6 |
| **#INT_IC7** | Input Capture #7 |
| **#INT_IC8** | Input Capture #8 |
| **#INT_LOWVOLT** | Low voltage detected |
| **#INT_LVD** | Low voltage detected |
| **#INT_MATHERR** | Arithmetic error trap |
| **#INT_MI2C** | Master I2C activity |
| **#INT_MI2C2** | Master2 I2C activity |
| **#INT_OC1** | Output Compare #1 |
| **#INT_OC2** | Output Compare #2 |
| **#INT_OC3** | Output Compare #3 |
| **#INT_OC4** | Output Compare #4 |
| **#INT_OC5** | Output Compare #5 |
| **#INT_OC6** | Output Compare #6 |
| **#INT_OC7** | Output Compare #7 |
| **#INT_OC8** | Output Compare #8 |
| **#INT_OSC_FAIL** | System oscillator failed |
| **#INT_PMP** | Parallel master port |
| **#INT_PMP2** | Parallel master port 2 |
| **#INT_PWM1** | PWM generator 1 time based interrupt |
| **#INT_PWM2** | PWM generator 2 time based interrupt |
| **#INT_PWM3** | PWM generator 3 time based interrupt |
| **#INT_PWM4** | PWM generator 4 time based interrupt |
| **#INT_PWMSEM** | PWM special event trigger |
| **#INT_QEI** | QEI position counter compare |
| **#INT_RDA** | RS232 receive data available |
| **#INT_RDA2** | RS232 receive data available in buffer 2 |

| | |
|---|---|
| **#INT_RTC** | Real - Time Clock/Calendar |
| **#INT_SI2C** | Slave I2C activity |
| **#INT_SI2C2** | Slave2 I2C activity |
| **#INT_SPI1** | SPI1 Transfer Done |
| **#INT_SPI1E** | SPI1E Transfer Done |
| **#INT_SPI2** | SPI2 Transfer Done |
| **#INT_SPI2E** | SPI2 Error |
| **#INT_SPIE** | SPI Error |
| **#INT_STACKERR** | Stack Error |
| **#INT_TBE** | RS232 transmit buffer empty |
| **#INT_TBE2** | RS232 transmit buffer 2 empty |
| **#INT_TIMER1** | Timer 1 overflow |
| **#INT_TIMER2** | Timer 2 overflow |
| **#INT_TIMER3** | Timer 3 overflow |
| **#INT_TIMER4** | Timer 4 overflow |
| **#INT_TIMER5** | Timer 5 overflow |
| **#INT_TIMER6** | Timer 6 overflow |
| **#INT_TIMER7** | Timer 7 overflow |
| **#INT_TIMER8** | Timer 8 overflow |
| **#INT_TIMER9** | Timer 9 overflow |
| **#INT_UART1E** | UART1 error |
| **#INT_UART2E** | UART2 error |
| **#INT_AUX** | Auxiliary memory ISR |

**Elements:**
*[PCD]* NOCLEAR, LEVEL=n, HIGH, FAST, ALT, CLR_FIRST

**Description:**
These directives specify the following function is an interrupt function. Interrupt functions may not have any parameters. Not all directives may be used with all parts. See the devices .h file for all valid interrupts for the part or in PCW use the pull down VIEW | Valid Ints

The compiler will generate code to jump to the function when the interrupt is detected. It will generate code to save and restore the machine state, and will clear the interrupt flag. To prevent the flag from being cleared add NOCLEAR after the #INT_xxxx. The application program must call ENABLE_INTERRUPTS(INT_xxxx) to initially activate the interrupt along with the ENABLE_INTERRUPTS(GLOBAL) to enable interrupts.

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

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

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

**Enable_interrupts -** specifies the levels that are enabled. The default is level 0 and level 7 is never disabled. High is the same as level = 7.

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

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

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

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

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

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

**A summary of the different kinds of PIC18 interrupts:**

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

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

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

**#INT_xxxx NOCLEAR** - The compiler will not clear the interrupt.

**#INT_xxx CLEAR_FIRST** - The compiler will clear the interrupt at the beginning of the ISR instead of the end. The user code in the function should call clear_interrput( ) to clear the interrupt in this case.

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

Some interrupts shown in the devices header file are only for the enable/disable interrupts. For example, **INT_RB3** may be used in enable/interrupts to enable pin B3. However, the interrupt handler is **#INT_RB**.

Similarly **INT_EXT_L2H** sets the interrupt edge to falling and the handler is **#INT_EXT**.

**Example Files:**
ex_sisr.c and ex_stwt.c

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

  #int_rtcc noclear
  isr(){
  ...
  }
  [PCD]
  #int_ad
  adc_handler(){
    adc_active=FALSE;
  }

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

**See Also:**
enable_interrupts(), disable_interrupts(), #INT_DEFAULT, #INT_GLOBAL, #PRIORITY

## #int_default

**Syntax:**
#int_default

**Elements:**
None

**Description:**
The following function will be called if the device 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.

*[PCD]* A #INT_xxx handler has not been defined for the interrupt.

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

**See Also:**
#INT_xxxx, #INT_global

## #int_global

**Syntax:**
#int_global

**Elements:**
None

**Description:**
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.

**Example Files:**
ex_glint.c

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

**See Also:**
#INT_xxxx

## __line__

**Syntax:**
__line__

**Elements:**
None

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

**Example Files:**
assert.h

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

**See Also:**
__file__

## #list

**Syntax:**
#list

**Elements:**
None

**Description:**
**#list** begins inserting or resumes inserting source lines into the **.lst** file after a **#NOLIST**.

**Example Files:**
16c74.h

**Examples:**
```
  #NOLIST               //Do not clutter up the list file
  #include<cdriver.h>
  #LIST
```

**See Also:**
#NOLIST

## #line

**Syntax:**
#line number file name

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

**Description:**
The C pre-processor informs the C Compiler of the location in your source code. This code is simply used to change the value of __LINE__ and __FILE__ variable.

**Examples:**

```
void main(){
#line 10        //specifies the line number that should be reported
                //for the following line of input

#line 7"hello.c" //line number in the source file hello.c and it sets
                 //the line 7 as current line and hello.c as current
file
```

## #locate

**Syntax:**
#locate ***id=x***

**Elements:**
**id** - is a C variable
**x** - is a constant memory address

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

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

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

**Example Files:**
ex_glint.c

**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
[PCD]
float x:
#locate x=0x800
```

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

## #module

**Syntax:**
#module

**Elements:**
None

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

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

**Examples:**

```
int GetCount(void);
void SetCount(int newCount);
#MODULE
int g_count;
#define G_COUNT_MAX  100
int GetCount(void) {return(g_count);}
void SetCount(int newCount) {
  if (newCount>G_COUNT_MAX)
     newCount=G_COUNT_MAX;
  g_count=newCount;
}
/*
```

```
the functions GetCount() and SetCount() have global scope, but the
variable g_count and the #define G_COUNT_MAX only has scope to this
file.
*/
```

**See Also:**
#EXPORT, Invoking the Command Line Compiler, Multiple Compilation Unit

## #nolist

**Syntax:**
#nolist

**Elements:**
None

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

**Example Files:**
16c74.h

**Examples:**
```
#NOLIST        //Do not clutter up the list list
#include<cdriver.h>
#LIST
```

**See Also:**
#LIST

## #ocs

**Syntax:**
#osc **x**

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

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

**Examples:**
```
#include<18F4520.h>
#device ICD=TRUE
#OCS 20 Mhz
```

```
#use rs232(debugger)

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

**See Also:**
#USE DELAY

## #opt

**Syntax:**
#opt **n**

**Elements:**
All Devices: ***n*** is the optimization level 1-9 or by using the word "compress" for PIC18 and Enhanced PIC16 families.
*[PCD]* All Devices: **n** is the optimization level 0-9

**Description:**
The optimization level is set with this directive. This setting applies to the entire program and may appear anywhere in the file. The default is 9 for normal. When Compress is specified the optimization is set to an extreme level that causes a very tight ROM image, the code is optimized for space, not speed. Debugging with this level my be more difficult.

**Examples:**
```
#opt5
```

## #org

**Syntax:**
#ORG ***start***, ***end***
or
#ORG ***segment***
or
#ORG ***start***, ***end { }***
or
#ORG ***start***, ***end auto=0***
#ORG ***start***,***end DEFAULT***
or
#ORG ***DEFAULT***

**Elements:**
***start -*** is the first ROM location (word address) to use.

***end -*** is the last ROM location.

***segment -*** is the start ROM location from a previous #ORG

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

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

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

If the keyword DEFAULT is used then this address range is used for all functions user and compiler generated from this point in the file until a #ORG DEFAULT is encountered (no address range). If a compiler function is called from the generated code while DEFAULT is in effect the compiler generates a new version of the function within the specified address range.
#ORG may be used to locate data in ROM. Because CONSTANT are implemented as functions the #ORG should proceed the CONSTANT and needs a start and end address. For a ROM declaration only the start address should be specified.

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

**Example Files:**
loader.c

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

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

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

#ORG 0x1B80
ROM int32 seridl_N0=12345;

#ORG 0x1C00, 0x1C0F               //This ID will be at 1C00
```

```
CHAR CONST ID[10}= {"123456789"}; //Note some extra code will
                                  //proceed the 123456789

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

**See Also:**
#ROM

## #pin_select

**Syntax:**
#PIN_SELECT **function=pin_xx**

**Elements:**
***function -*** is the Microchip defined pin function name, such as:

- U1RX(UART1 receive)
- INT1(external interrupt 1)
- T2CK (timer 2 clock)
- IC1 (input capture 1)
- OC1 (output capture 1)

**PCB, PCM, PCH**

| **INT1** | **External Interrupt 1** |
|---|---|
| **INT2** | External Interrupt 2 |
| **INT3** | External Interrupt 3 |
| **T0CK** | Timer0 External Clock |
| **T3CK** | Timer3 External Clock |
| **CCP1** | Input Capture 1 |
| **CCP2** | Input Capture 2 |
| **T1G** | Timer1 Gate Input |
| **T3G** | Timer3 Gate Input |
| **U2RX** | EUSART2 Asynchronous Receive/Synchronous Receive (also named: RX2) |
| **U2CK** | EUSART2 Asynchronous Clock Input |
| **SDI2** | SPI2 Data Input |
| **SCK2IN** | SPI2 Clock Input |
| **SS2IN** | SPI2 Slave Select Input |
| **FLT0** | PWM Fault Input |
| **T0CKI** | Timer0 External Clock Input |
| **T3CKI** | Timer3 External Clock Input |
| **RX2** | EUSART2 Asynchronous Transmit/Asynchronous Clock Output (also named: TX2) |
| **NULL** | NULL |

| | |
|---|---|
| **C1OUT** | Comparator 1 Output |
| **C2OUT** | Comparator 2 Output |
| **U2TX** | EUSART2 Asynchronous Transmit/ Asynchronous Clock Output (also named: TX2) |
| **U2DT** | EUSART2 Synchronous Transmit (also named: DT2) |
| **SDO2** | SPI2 Data Output |
| **SCK2OUT** | SPIC2 Clock Output |
| **SS2OUT** | SPI2 Slave Select Output |
| **ULPOUT** | Ultra Low-Power Wake-Up Event |
| **P1A** | ECCP1 Compare or PWM Output Channel A |
| **P1B** | ECCP1 Enhanced PWM Output, Channel B |
| **P1C** | ECCP1 Enhanced PWM Output, Channel C |
| **P1D** | ECCP1 Enhanced PWM Output, Channel D |
| **P2A** | ECCP2 Compare or PWM Output Channel A |
| **P2B** | ECCP2 Enhanced PWM Output, Channel B |
| **P2C** | ECCP2 Enhanced PWM Output, Channel C |
| **P2D** | ECCP1 Enhanced PWM Output, Channel D |
| **TX2** | EUSART2 Asynchronous Transmit/Asynchronous Clock Output (also named: TX2) |
| **DT2** | EUSART2 Synchronous Transmit (also named: U2DT) |
| **SCK2** | SPI2 Clock Output |
| **SSDMA** | SPI DMA Slave Select |

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

**PCD (PIC24/dsPIC devices)**

| **NULL** | **NULL** |
|---|---|
| **C1OUT** | Comparator 1 Output |
| **C2OUT** | Comparator 2 Output |
| **C3OUT** | Comparator 3 Output |
| **C4OUT** | Comparator 4 Output |
| **U1TX** | UART1 Transmit |
| **U1RTS** | UART1 Request to Send |
| **U2TX** | UART2 Transmit |
| **U2RTS** | UART2 Request to Send |
| **U3TX** | UART3 Transmit |
| **U3RTS** | UART3 Request to Send |
| **U4TX** | UART4 Transmit |
| **U4RTS** | UART4 Request to Send |
| **SDO1** | SPI1 Data Output |
| **SCK1OUT** | SPI1 Clock Output |
| **SS1OUT** | SPI1 Slave Select Output |
| **SDO2** | SPI2 Data Output |
| **SCK2OUT** | SPI2 Clock Output |
| **SS2OUT** | SPI2 Slave Select Output |
| **SDO3** | SPI3 Data Output |

| | |
|---|---|
| **SCK3OUT** | SPI3 Clock Output |
| **SS3OUT** | SPI3 Slave Select Output |
| **SDO4** | SPI4 Data Output |
| **SCK4OUT** | SPI4 Clock Output |
| **SS4OUT** | SPI4 Slave Select Output |
| **OC1** | Output Compare 1 |
| **OC2** | Output Compare 2 |
| **OC3** | Output Compare 3 |
| **OC4** | Output Compare 4 |
| **OC5** | Output Compare 5 |
| **OC6** | Output Compare 6 |
| **OC7** | Output Compare 7 |
| **OC8** | Output Compare 8 |
| **OC9** | Output Compare 9 |
| **OC10** | Output Compare 10 |
| **OC11** | Output Compare 11 |
| **OC12** | Output Compare 12 |
| **OC13** | Output Compare 13 |
| **OC14** | Output Compare 14 |
| **OC15** | Output Compare 15 |
| **OC16** | Output Compare 16 |
| **C1TX** | CAN1 Transmit |
| **C2TX** | CAN2 Transmit |
| **CSDO** | DCI Serial Data Output |
| **CSCKOUT** | DCI Serial Clock Output |
| **COFSOUT** | DCI Frame Sync Output |
| **UPDN1** | QEI1 Direction Status Output |
| **UPDN2** | QEI2 Direction Status Output |
| **CTPLS** | CTMU Output Pulse |
| **SYNCO1** | PWM Synchronization Output Signal |
| **SYNCO2** | PWM Secondary Synchronization Output Signal |
| **REFCLKO** | REFCLK Output Signal |
| **CMP1** | Analog Comparator Output 1 |
| **CMP2** | Analog Comparator Output 2 |
| **CMP3** | Analog Comparator Output 3 |
| **CMP4** | Analog Comparator Output 4 |
| **PWM4H** | PWM4 High Output |
| **PWM4L** | PWM4 Low Output |
| **QEI1CCMP** | QEI1 Counter Comparator Output |
| **QEI2CCMP** | QEI2 Counter Comparator Output |
| **MDOUT** | DSM Modulator Output |
| **DCIDO** | DCI Serial Data Output |
| **DCISCKOUT** | DCI Serial Clock Output |
| **DCIFSOUT** | DCI Frame Sync Output |
| **INT1** | External Interrupt 1 Input |

| | |
|---|---|
| **INT2** | External Interrupt 2 Input |
| **INT3** | External Interrupt 3 Input |
| **INT4** | External Interrupt 4 Input |
| **T1CK** | Timer 1 External Clock Input |
| **T2CK** | Timer 2 External Clock Input |
| **T3CK** | Timer 3 External Clock Input |
| **T4CK** | Timer 4 External Clock Input |
| **T5CK** | Timer 5 External Clock Input |
| **T6CK** | Timer 6 External Clock Input |
| **T7CK** | Timer 7 External Clock Input |
| **T8CK** | Timer 8 External Clock Input |
| **T9CK** | Timer 9 External Clock Input |
| **IC1** | Input Capture 1 |
| **IC2** | Input Capture 2 |
| **IC3** | Input Capture 3 |
| **IC4** | Input Capture 4 |
| **IC5** | Input Capture 5 |
| **IC6** | Input Capture 6 |
| **IC7** | Input Capture 7 |
| **IC8** | Input Capture 8 |
| **IC9** | Input Capture 9 |
| **IC10** | Input Capture 10 |
| **IC11** | Input Capture 11 |
| **IC12** | Input Capture 12 |
| **IC13** | Input Capture 13 |
| **IC14** | Input Capture 14 |
| **IC15** | Input Capture 15 |
| **IC16** | Input Capture 16 |
| **C1RX** | CAN1 Receive |
| **C2RX** | CAN2 Receive |
| **OCFA** | Output Compare Fault A Input |
| **OCFB** | Output Compare Fault B Input |
| **OCFC** | Output Compare Fault C Input |
| **U1RX** | UART1 Receive |
| **U1CTS** | UART1 Clear to Send |
| **U2RX** | UART2 Receive |
| **U2CTS** | UART2 Clear to Send |
| **U3RX** | UART3 Receive |
| **U3CTS** | UART3 Clear to Send |
| **U4RX** | UART4 Receive |
| **U4CTS** | UART4 Clear to Send |
| **SDI1** | SPI1 Data Input |
| **SCK1IN** | SPI1 Clock Input |
| **SS1IN** | SPI1 Slave Select Input |
| **SDI2** | SPI2 Data Input |

| | |
|---|---|
| **SCK2IN** | SPI2 Clock Input |
| **SS2IN** | SPI2 Slave Select Input |
| **SDI3** | SPI3 Data Input |
| **SCK3IN** | SPI3 Clock Input |
| **SS3IN** | SPI3 Slave Select Input |
| **SDI4** | SPI4 Data Input |
| **SCK4IN** | SPI4 Clock Input |
| **SS4IN** | SPI4 Slave Select Input |
| **CSDI** | DCI Serial Data Input |
| **CSCK** | DCI Serial Clock Input |
| **COFS** | DCI Frame Sync Input |
| **FLTA1** | PWM1 Fault Input |
| **FLTA2** | PWM2 Fault Input |
| **QEA1** | QEI1 Phase A Input |
| **QEA2** | QEI2 Phase A Input |
| **QEB1** | QEI1 Phase B Input |
| **QEB2** | QEI2 Phase B Input |
| **INDX1** | QEI1 Index Input |
| **INDX2** | QEI2 Index Input |
| **HOME1** | QEI1 Home Input |
| **HOME2** | QEI2 Home Input |
| **FLT1** | PWM1 Fault Input |
| **FLT2** | PWM2 Fault Input |
| **FLT3** | PWM3 Fault Input |
| **FLT4** | PWM4 Fault Input |
| **FLT5** | PWM5 Fault Input |
| **FLT6** | PWM6 Fault Input |
| **FLT7** | PWM7 Fault Input |
| **FLT8** | PWM8 Fault Input |
| **SYNCI1** | PWM Synchronization Input 1 |
| **SYNCI2** | PWM Synchronization Input 2 |
| **DCIDI** | DCI Serial Data Input |
| **DCISCKIN** | DCI Serial Clock Input |
| **DCIFSIN** | DCI Frame Sync Input |
| **DTCMP1** | PWM Dead Time Compensation 1 Input |
| **DTCMP2** | PWM Dead Time Compensation 2 Input |
| **DTCMP3** | PWM Dead Time Compensation 3 Input |
| **DTCMP4** | PWM Dead Time Compensation 4 Input |
| **DTCMP5** | PWM Dead Time Compensation 5 Input |
| **DTCMP6** | PWM Dead Time Compensation 6 Input |
| **DTCMP7** | PWM Dead Time Compensation 7 Input |

**Description:**
When using PPS chips a #PIN_SELECT must be appear before these peripherals can be used or referenced.

*[PCD]*On devices that contain Peripheral Pin Select (PPS), this allows the programmer to define which pin a peripheral is mapped to.

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

**See Also:**
pin_select()

## __pcb__

**Syntax:**
__pcb__

**Elements:**
None

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

**Example Files:**
ex_sqw.c

**Examples:**
```
#ifdef __pcb__
#device PIC16C54
#endif
```

**See Also:**
__PCM__, __PCH__, __PCD__

## __pcd__

**Syntax:**
__pcd__

**Elements:**
None

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

**Example Files:**
ex_sqw.c

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

**See Also:**
__PCB__, __PCM__, __PCH__

## __pcm__

**Syntax:**
__pcm__

**Elements:**
None

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

**Example Files:**
ex_sqw.c

**Examples:**
```
  #ifdef __pcm__
  #device PIC16C71
  #endif
```

**See Also:**
__PCB__, __PCH__, __PCD__

## __pch__

**Syntax:**
__pch__

**Elements:**
None

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

**Example Files:**
ex_sqw.c

**Examples:**
```
#ifdef __pch__
#device PIC18F452
#endif
```

**See Also:**
__PCM__, __PCM__, __PCD__

## #pragma

**Syntax:**
#pragma **cmd**

**Elements:**
**cmd** - is any valid pre-processor directive.

**Description:**
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.

**Example Files:**
ex_cust.c

**Examples:**
```
#pragma device PIC16C54
```

**See Also:**

## #priority

**Syntax:**
#priority **ints**

**Elements:**
**ints** - is a list of one or more interrupts separated by commas.

**exports** - makes the functions generated from this directive available to other compilation units within the link.

**Description:**
The priority directive may be used to set the interrupt priority. The highest priority items are first in the list. If an interrupt is active it is never interrupted. If two interrupts occur at around the same time then the higher one in this list will be serviced first. When linking multiple compilation units be aware only the one in the last compilation unit is used.

**Examples:**
```
#priority rtc_c.rb
```

**See Also:**
#INT_xxxx

## #profile

**Syntax:**
#profile **options**

**Elements:**
**options** - may be one of the following:

**functions -** Profiles the start/end of functions and all profileout() messages.

**functions, parameters -** Profiles the start/end of functions, parameters sent to functions, and all profileout() messages.

**profileout -** Only profile profileout() messages.

**paths -** Profiles every branch code.

**off -** Disable all code profiling.

**on -** Re-enables the code profiling that was previously disabled with a #profile off command. This will use the last options before disabled with the off command.

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

**Example Files:**
ex_profile.c

**Examples:**

```
#profile off
void BigFunction(void)
{
//BigFunction code goes here since #profile off was called above.
//No profiling will happen even for the functions called by
BigFunction().
}
#profile on
```

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

## #recursive

**Syntax:**
#recursive

**Elements:**
None

**Description:**
Directs the compiler that the procedure immediately following the directive will be recursive.

**Examples:**

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

## #reserve

**Syntax:**
#reserve ***address***
#reserve ***address, address, address***
#reserve ***start:end***

**Elements:**
**address** - is a RAM address.

**start -** is the first address.

**end -** is the last address.

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

**Example Files:**
ex_cust.c

**Examples:**
```
#device PIC16C74
#reserve 0x60:0X6f

[PCD]
#device dsPIC30F2010
#reserve 0x800:0x80B3
```

**See Also:**
#ORG

## #rom

**Syntax:**
#rom ***address*** = {**list**}

**Elements:**
**address** - is the same address used in the device datasheet (Byte for PIC18 and Word for all others)

**list -** is a list of words separated by commas.

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

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

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

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

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

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

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

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

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

**Example Files:**
ex_glint.c

**Examples:**

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

**See Also:**
#ORG

## #separate

**Syntax:**
#separate
*[PCD]* #separate ***options***

**Elements:**
*[PCD]* **options** - options include:

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

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

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

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

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

Use STDCALL with the ARG, DND or AVOID parameters.

If one of these options is not specified, the compiler will determine the best configuration, and will usually not use the stack for function parameters (usually scratch space is allocated for parameters).

**Description:**
Directs 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*****,** separated as requested, even if there is not enough stack space to execute.

**Example Files:**
ex_cust.c

**Examples:**

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

**See Also:**
#INLINE

## #serialize

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

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

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

Use in place of ***id*** parameter, when storing serial number to EEPROM:
***dataee=x*** - The address x is the start address in the data EEPROM.
***binary=x*** - The integer x is the number of bytes to be written to address specified.
***string=x*** - The integer x is the number of bytes to be written to address specified.
***unicode=n -*** If n is a 0, the string format is normal unicode. For n>0 n indicates the string
number in a USB descriptor.

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

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

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

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

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

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

**Examples:**
```
//Prompt user for serial number to be placed
//at address of serialNumA
//Default serial number = 200int8int8 const serialNumA=100;
//#serialize(id=serialNumA,next="200",prompt="Enter the serial number")

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

```
//log="seriallog.txt")

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

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

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

## #task

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

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

**Syntax:**
#task ***(options)***

**Elements:**
***options*** are separated by comma and may be:

**rate=time** - Where time is a number followed by s, ms, us, or ns. This specifies how often the task will execute.

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

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

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

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

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

The max option is used to specify how much processor time a task will use in one execution of the task. The time specified in max must be equal to or less than the time

specified in the minor_cycle option of the #USE RTOS directive before the project will compile successfully. The compiler does not have a way to enforce this limit on processor time, so a programmer must be careful with how much processor time a task uses for execution. This option does not need to be specified.

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

**Examples:**

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

**See Also:**
#USE RTOS

## __time__

**Syntax:**
__time__

**Elements:**
None

**Description:**
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__);
```

## #todo

**Syntax:**
#todo ***text***

**Elements:**
***text*** is free text

**Description:**
This directive documents in the source code items that the developer needs to work on.

**Example Files:**
None

**Examples:**

```
#todo Verify the math works in convert_adc_values
```

**See Also:**
PCW Overview

## #type

**Syntax:**
#TYPE ***standard-type*=*size***

#TYPE ***default=area***

#TYPE **unsigned**

#TYPE **signed**

*[PCD]* #TYPE **char=signed**

*[PCD]* #TYPE **char=unsigned**

*[PCD]* #TYPE ARG=Wx:Wy

*[PCD]* #TYPE DND=Wx:Wy

*[PCD]* #TYPE AVOID=Wx:Wy

*[PCD]* #TYPE RECURSIVE

*[PCD]* #TYPE CLASSIC

**Elements:**
***standard-type* -** is one of the C keywords short, int, long, or default

*[PCD]* s***tandard-type* -** is one of the C keywords short, int, long, float, or double

***size* -** is 1,8,16, or 32

*[PCD]* ***size* -** is 1,8,16, 48, or 64

***area* -** is a memory region defined before the #TYPE using the addressmod directive

*[PCD]* **Wx:Wy -** is a range of working registers (example: W0, W1, W15, etc)

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

Note that the commas are optional. 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.
**Note:** CCS example programs and include files may not work correctly when using #TYPE in the program.

*[PCD]* Classic will set the type sizes to be compatible with CCS PIC programs.

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

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

The #TYPE directive allows the keywords UNSIGNED and SIGNED to set the default data type.

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

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

**Example Files:**
ex_cust.c

**Examples:**
```
#TYPE   SHORT= 8, INT= 16, LONG= 32

#TYPE default=area

addressmod (user_ram_block, 0x100, 0x1FF);

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

#type default=                // restores memory allocation
```

```
                                // back to normal

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

[PCD]
#TYPE   SHORT=1, INT=8, LONG=16, FLOAT=48

#TYPE default=area

addressmod (user_ram_block, 0x100, 0x1FF);

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

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

#TYPE SIGNED

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

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

## #undef

**Syntax:**
#undef ***id***

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

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

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

**See Also:**
#DEFINE

## __unicode__

**Syntax:**
__unicode(constant-string)

**Elements:**
Unicode format string

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

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

Since the normal C terminator is not used for these strings you need to do one of the following for variable length strings:

```
string = _unicode(KEYWORD)  "\000\000";
```
OR
```
string = _unicode(KEYWORD);
string_size = sizeof(_unicode(KEYWORD));
```

**Example Files:**
usb_desc_hid.h

**Examples:**
```
#define USB_DESC_STRING_TYPE 3

#define USB_STRING(x) (sizeof(_unicode(x))+2), USB_DESC_STRING_TYPE,
__unicode(x)
#define USB_ENGLISH_STRING 4,USB_DESC_STRING_TYPE,0x09,0x04
                                          //Microsoft defined for US
English

char const USB_STRING_DESC[]=[
```

```
   USB_ENGLISH_STRING;
   USB_STRING("CCS");
   USB_STRING("CCS HID DEMO")
};
```

## #use capture

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

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

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

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

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

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

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

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

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

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

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

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

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

**DEFINE=id -** Creates a define named id which specifies the number of capture per second. Default define name if not specified is CAPTURES_PER_SECOND. Define name must start with an ASCII letter 'A' to 'Z', an ASCII letter 'a' to 'z' or an ASCII underscore ('_').

**Description:**
This directive tells the compiler to setup an input capture on the specified pin using the specified settings. The #USE DELAY directive must appear before this directive can be used. This directive enables use of built-in functions such as get_capture_time() and get_capture_event().

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

**See Also:**
get_capture_time(), get_capture_event()

## #use_delay

**Syntax:**
#use_delay **(options)**

**Elements:**
**Options -** may be any of the following separated by commas:
***clock=speed*** speed is a constant 1-100000000 (1 hz to 100 mhz).

This number can contains commas. This number also supports the following denominations: M, MHZ, K, KHZ. This specifies the clock the CPU runs at. Depending on the PIC this is 2 or 4 times the instruction rate. This directive is not needed if the following type=speed is used and there is no frequency multiplication or division.

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

***restart_wdt*** will restart the watchdog timer on every delay_us() and delay_ms() use.

***clock_out*** when used with the internal or oscillator types this enables the clockout pin to output the clock.

***fast_start*** some chips allow the chip to begin execution using an internal clock until the primary clock is stable.

***lock*** some chips can prevent the oscillator type from being changed at run time by the software.

***USB or USB_FULL*** for devices with a built-in USB peripheral. When used with the ***type=speed*** option the compiler will set the correct configuration bits for the USB peripheral to operate at Full-Speed.

***USB_LOW*** for devices with a built-in USB peripheral. When used with the ***type=speed*** option the compiler will set the correct configuration bits for the USB peripheral to operate at Low-Speed.

**PLL_WAIT** for devices with a PLL and a PLL Ready Status flag to test. When a PLL clock is specified it will cause the compiler to poll the ready PLL Ready Flag and only continue program execution when flag indicates that the PLL is ready.

**ACT or ACT=type** for device with Active Clock Tuning, type can be either USB or SOSC. If only using ACT type will default to USB. ACT=USB causes the compiler to enable the active clock tuning and to tune the internal oscillator to the USB clock. ACT=SOSC causes the compiler to enable the active clock tuning and to tune the internal oscillator to the secondary clock at 32.768 kHz. ACT can only be used when the system clock is set to run from the internal oscillator.

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

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

**Description:**
Tells the compiler the speed of the processor and enables the use of the built-in functions: delay_ms() and delay_us(). Will also set the proper configuration bits, and if needed configure the internal oscillator. Speed is in cycles per second. An optional restart_wdt may be used to cause the compiler to restart the WDT while delaying. When linking multiple compilation units, this directive must appear in any unit that needs timing configured (delay_ms(), delay_us(), UART, SPI).

In multiple clock speed applications, this directive may be used more than once. Any timing routines (delay_ms(), delay_us(), UART, SPI) that need timing information will use the last defined #USE DELAY (For initialization purposes, the compiler will initialize the configuration bits and internal oscillator based upon the first #USE DELAY.

**Example Files:**
ex_sqw.c

**Examples:**

```
                     // set timing config to 32KHz, User sets the fuses
                    // on delay_us() and delay_ms()
#use delay (clock=32000, RESTART_WDT)

                   //the following 4 examples all configure the timing
library
                   //to use a 20Mhz clock, where the source is a
crystal.
#use delay (crystal=20000000)
#use delay (xtal=20,000,000)
#use delay(crystal=20Mhz)
#use delay(clock=20M, crystal)

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

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

**See Also:**
delay_ms(), delay_us()

## #use dynamic_memory

**Syntax:**
#use dynamic_memory

**Elements:**
None

**Description:**
This pre-processor directive instructs the compiler to create the _DYNAMIC_HEAD object. _DYNAMIC_HEA is the loation where the first free space is allocated.

**Example Files:**
ex_malloc.c

**Examples:**
```
#USE DYNAMIC_MEMORY
void main(){
}
```

## #use fast_io

**Syntax:**
#use fast_io **(port)**

**Elements:**
**port** - is A, B, C, D, E, F, G, H, J or ALL

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

**Example Files:**
ex_cust.c

**Examples:**
```
#use fast_io(A)
```

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

## #use fixed_io

**Syntax:**
#use fixed_io **(port_outputs=pin, pin?)**

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

**Description:**
This directive affects how the compiler will generate code for input and output instructions that follow. This directive takes effect until another #USE XXX_IO directive is encountered. The fixed method of doing I/O will cause the compiler to generate code to make an I/O pin either input or output every time it is used. The pins are programmed

according to the information in this directive (not the operations actually performed). This saves a byte of RAM used in standard I/O. When linking multiple compilation units be aware this directive only applies to the current compilation unit.

**Examples:**

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

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

## #use i2c

**Syntax:**
#use i2c **(options)**

**Elements:**
**options** - are separated by commas and may include the following:

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

| | |
|---|---|
| **I2C1** | Instead of SCL= and SDA= this sets the pins to the first module |
| **I2C2** | Instead of SCL= and SDA= this sets the pins to the second module |
| **NOINIT** | No initialization of the I2C peripheral is performed. Use I2C_INIT() to initialize peripheral at run time. |

Only some chips allow the following:

| | |
|---|---|
| **DATA_HOLD** | **No ACK is sent until I2C_READ is called for data bytes (slave only)** |
| **ADDRESS_HOLD** | No ACK is sent until I2C_read is called for the address byte (slave only) |
| **SDA_HOLD** | Min of 300ns holdtime on SDA a from SCL goes low |

PIC18 devices that have a separate I2C peripheral instead of a combined SSP peripheral allow the following:

| | |
|---|---|
| **CLOCK_SOURCE=x** | **Used to specify the I2C peripheral's clock source. Options can be FOSC/4, FOSC, HFINTOSC or HFINT, MFINTOSC or MFINT, REFCLK or REF, TIMER0 or TMR0, TIMER2 or TMR2, TIMER4 or TMR4, TIMER6 or TMR6, or SMT1. If not specified, it defaults to FOSC/4. If a peripheral is selected as the clock, TMR2 for example, that peripheral must be setup to achieve the desired I2C clock rate.** |
| **CLOCK_DIVISOR=x** | Used to specify the I2C clock divisor, can be 4 or 5. |
| **ADDRESS_BITS=x** | Used to specify the number of address bits, can be set to 7 or 10. Default is 7, if not specified. |
| **ADDRESS1=x**<br>**ADDRESS2=x**<br>**ADDRESS3=x**<br>**ADDRESS4=x** | Used to specify the slave mode addresses the peripheral will respond to. Depending on the address bits and number of address masks, these devices can have 4, 2 or 1 addresses that they will respond to. These allow setting the individual addresses, ADDRESS1=x is the same as ADDRESS=xx.<br>When set for 7 bit address mode, can have 4 or 2 addresses. 4 if no masks are specified. 2 if 1 or 2 masks are specified.<br>When set for 10 bit address mode, can have 2 or 1 addresses. 2 if no masks are specified and 1 if 1 mask is specified. If using 1 or more addresses, always start with ADDRESS1, then ADDRESS2, etc., because any unspecified addresses will be assigned to the value of ADDRESS1. |
| **MASK1=x**<br>**MASK2=x** | Used to specify the slave mode address masks, depending on the number of address bits it can have, 0, 1 or 2 address masks. When set for 7 bit address mode, it can have 0 or 2 address masks; simply assigning a value to MASK1 means I2C peripheral is set for 2 addresses and 2 address masks. When |

| | |
|---|---|
| | set for 10 bit address mode, it can have 0 or 1 address masks; simply assigning a value to MASK1 means the peripheral is set for 1 address and 1 address mask. If using 1 or more address mask, always set MASK1 because if 7 bit address mode is used and only MASK1 is specified, both address masks will be set to the value of MASK1. When using 7 bit address mode, MASK1 is the mask for ADDRESS1 and MASK2 is the mask for ADDRESS2. |
| **SDA_HOLD=x** | Used to set the hold time on SDA after falling edge of SCL, can be set to 30, 100 or 300ns. If only SDA_HOLD is specified, the hold time is set to 300ns and if not specified, the hold time is set to 100ns. |

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

The I2C library contains functions to implement an I2C bus. The #USE I2C remains in effect for the I2C_START, I2C_STOP, I2C_READ, I2C_WRITE and I2C_POLL functions until another USE I2C is encountered. If hardware pins are specified for SDA and SCL, then hardware functions are generated unless the **force_sw** is specified; otherwise software functions are generated. The SLAVE mode should only be used with the built-in SSP. The functions created with this directive are exported when using multiple compilation units. To access the correct function use the stream identifier.
*[PCD]* Some devices have an alternate set of I2C pins that may be used with the hardware I2C peripherals instead of the default pins. If a device has alternative I2C pins, then they will have the following configuration fuses available for selecting which pair to use: ALTI2Cx and NOALTI2Cx. x being the I2C peripheral (1-3). Setting the NOALTI2Cx configuration fuse causes the device to use the ASCLx and ASDAx pins for the peripheral. Additionally, these configuration fuses determine which pins #use i2c() determines the hardware I2C pins for each I2C peripheral. By default, the NOALTI2Cx configuration fuses are set. In order to use the alternative I2C hardware pins, the ALTI2Cx configuration fuse must be set for that I2C peripheral.

**Example Files:**
ex_extee.c with 16c74.h

**Examples:**
```
  #use i2c(master, sda-PIN_B0, sci=PIN_B1

  #use i2c(slave, sda=PIN_C4, sci=PIN_C3
    address=0xa0,FORCE_HW
```

```
#use i2c(master, sci=PIN_B0, sda=PIN_B1, fast=450000)
                    //sets the target speed to 450 KBSP
```

**See Also:**
i2c_poll, i2c_speed, i2c_start, i2c_stop, i2c_slaveaddr, i2c_isr_state, i2c_write, i2c_read, I2C Overview

## #use profile()

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

**Elements:**
**option** - may be any of the following separated by a comma:

**ICD** - (Default) configures code profiler to use the ICD connection.

**TIMER1** - (optional) if specified, the code profiler run-time on the microcontroller will use the Timer1 peripheral as a timestamp for all profile events. If not specified, the code profiler tool will use the PC clock, which may be accurate for fast events.

**BAUD=x -** (optional) if specified, will use a different baud rate between the microcontroller and the code profiler tool. This may be required on slow microcontrollers to attempt to use a slower baud rate.

**Description:**
This directs the compiler to add the code profiler run-time in the microcontroller and configure the link and clock.

**Example Files:**
ex_profile.c

**Examples:**

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

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

## #use pwm()

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

**Elements:**
**option** - may be any of the following separated by a comma:

**PWMx or CCPx** - Selects the CCP to use, **x** being the module to use.

*[PCD]* **PWMx or OCx -** Selects the Output Compare module, **x** being the module number to use.

**OUTPUT=PIN_xx -** Selects the PWM pin to use, pin must be one of the CCP pins. If device has remappable pins compiler will assign specified pin to specified CCP module. If CCP module not specified it will assign remappable pin to first available module.

*[PCD]* **OUTPUT=PIN_xx -** Selects the PWM pin to use, pin must be one of the OC pins. If device has remappable pins compiler will assign specified pin to specified OC module. If OC module not specified it will assign remappable pin to first available module.

**TIMER=x -** Selects timer to use the PWM module, default if not specified is Timer2.

**FREQUENCY=x -** Sets the period of PWM based off specified value, should not be used if PERIOD is already specified. If frequency can't be achieved exactly compiler will generate a message specifying the exact frequency and period of PWM. If neither FREQUENCY or PERIOD is specified, the period defaults to maximum possible period with maximum resolution and compiler will generate a message specifying the frequency and period of PWM, or if using same timer as previous stream instead of setting to maximum possible it will be set to the same as previous stream. If using same timer as previous stream and frequency is different compiler will generate an error.

**Period=x -** Sets the period of PWM, should not be used if FREQUENCY is already specified. If period can't be achieved exactly compiler will generate a message specifying the exact period and frequency of PWM. If neither PERIOD or FREQUENCY is specified, the period defaults to maximum possible period with maximum resolution and compiler will generate a message specifying the frequency and period of PWM, or if using same timer as previous stream instead of setting to maximum possible it will be set to the same as previous stream. If using same timer as previous stream and period is different compiler will generate an error.

**BITS=x -** Sets the resolution of the the duty cycle, if period or frequency is specified will adjust the period to meet set resolution and will generate an message specifying the frequency and duty of PWM. If period or frequency not specified will set period to maximum possible for specified resolution and compiler will generate a message specifying the frequency and period of PWM, unless using same timer as previous then it will generate an error if resolution is different then previous stream. If not specified, then frequency, period or previous stream using same timer sets the resolution.

**DUTY=x -** Selects the duty percentage of PWM. Default, if not specified, is 50%.

**PWM_ON -** Initialize the PWM in the ON state. Default state, if not specified, is ***pwm_on*** or ***pwm_off***.

**PWM_OFF -** Initialize the PWM in the OFF state.

**STEAM=id -** Associates a stream identifier with the PWM signal. The identifier may be used in functions like ***pwm_set_duty_percent()***.

**Description:**
This directive tells the compiler to setup a PWM on the specified pin using the specified frequency, period, duty cycle and resolution. The #USE DELAY directive must appear before this directive can be used. This directive enables use of built-in functions such as ***pwm_set_duty_percent()***, ***pwm_set_frequency()***, ***pwm_set_duty()***, ***pwm_on()*** and ***pwm_off()***.

**See Also:**
pwm_on(), pwm_off(), pwm_set_frequency(), pwm_set_duty_percent(), pwm_set_duty()

## #use rs232

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

**Elements:**
**option** - may be any of the following separated by a comma:

**STREAM=id** - Associates a stream identifier with this RS232 port. The identifier may then be used in functions like fputc.

**BAUD=x -** Set baud rate to **x**.

**XMIT=pin -** Set transmit pin.

**RCV=pin -** Set receive pin.

**FORCE_SW -** Generate software serial I/O routines even when UART pins are specified.

**BRGH10K -** Allow bad baud rates on devices that have baud rate problems.

**ENABLE=pin -** The specified pin will be high during transmit. This may be used to enable 485 transmit.

**DEBUGGER -** Indicates this stream is used to send/receive data through a CCS ICD unit. The default pin used is B3, use **XMIT=** and **RCV=** to change the pain used. Both should be the same pin.

**RESTART_WDT -** Causes 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 MAX232). May not be used with internal UART.

**PARITY=x -** Where **x** is N, E, or O.

**BITS=x -** Where **x** is 5-9 (5-7 may not be used with the SCI).

**FLOAT_HIGH -** The line is not driven high. This is used for open collector outputs. Bit 6 in RS232_ERRORS is set if the pin is not high at the end of the bit time.

**ERRORS -** Used for the compiler to keep receive errors in the variable RS232_ERRORS and to reset errors when they occur, RS232_BUFFER_ERRORS when transmit, and RECEIVE_BUFFER are used.

**SAMPLE_EARLY -** A getc() normally samples data in the middle of a bit time. This option causes the sample to be at the start of a bit time. May not be used with UART.

**RETURN=pin -** The pin used to read signal back for **FLOAT_HIGH** and **MULTI_MASTER**. The default for **FLOAT_HIGH** is the **XMIT** pin, and for **MULTI_MASTER** the **RCV** pin.

**MULTI_MASTER -** Uses the RETURN pin to determine if another master on the bus is transmitting at the same time. If a collision is detected bit 6 is set in RS232_ERRORS and all future PUTC's are ignored until bit 6 is cleared. The signal is checked at the start and end of a bit time. May not be used with the UART.

**LONG_DATA -** Makes getc() return an int16 and putc() accept an int16. This is for 9 bit data formats.

**DISABLE_INTS -** Will cause interrupts to be disabled when the routines get or put a character. This prevents character distortion for software implemented I/O and prevents interaction between I/O in interrupt handlers and the main program when using the UART.

**STOP=x -** Used to set the number of stop bits (default is 1). This works for both UART and non-UART ports.

**TIMEOUT=x -** To set the time getc() waits for a byte in milliseconds. If no character comes in within this time the RS232_ERRORS is set to 0 as well as the return value form getc(). This works for both UART and non-UART ports.

**SYNC_SLAVE -** Makes the RS232 line a synchronous slave, making the receive pin a clock in, and the data pin the data in/out.

**SYNC_MASTER -** Makes the RS232 line a synchronous master, making the receive pin a clock out, and the data pin the data in/out.

**SYNC_MASTER_CONT -** Makes the RS232 line a synchronous master mode in continuous receive mode. The receive pin is set as a clock out, and the data pin is set as the data in/out.

**UART1 -** Sets the XMIT= and RCV= to the device's first hardware UART.

**UART2 -** Sets the XMIT= and RCV= to the chips second hardware UART.

**UART3 -** Sets the XMIT= and RCV= to the chips third hardware UART.

**UART4 -** Sets the XMIT= and RCV= to the chips fourth hardware UART.

*[PCD]* **UART1A -** Uses alternate UART pins.

*[PCD]* **UART2A -** Uses alternate UART pins.

**NOINIT -** No initialization of the UART peripheral is performed. Useful for dynamic control of the UART baud rate or initializing the peripheral manually at a later point in the program's run time. If this option is used, then setup_uart( ) needs to be used to initialize the peripheral. Using a serial routine (such as getc( ) or putc( )) before the UART is initialized will cause undefined behavior.

**ICD -** Indicates this stream uses the ICD in a special pass through mode to send/receive serial data to/from the PC. The ICSP clock line is the device's receive pin (usually B6), and the ICSP data line is the transmit pin (usually B7). The default transmit pin is the device's ICSPDAT/PGD pin and the default receive pin is the device's ICSPCLK/PGC pin. Use XMIT= and RCV= to change the pins used.

**MAX_ERROR=x -** Specifies the max error percentage the compiler can set the RS232 baud rate from the specified baud before generating an error. Defaults to 3% if not specified.

**serial buffer options:**

**RECEIVE_BUFFER=x -** Size in bytes of UART circular receive buffer, default if not specified is zero. Uses an interrupt to receive data, supports RDA interrupt or external interrupts.

**TRANSMIT_BUFFER=x -** Size in bytes of UART circular transmit buffer, default if not specified is zero.

**TXISR -** If TRANSMIT_BUFFER is greater then zero specifies using TBE interrupt for transmitting data. Default is NOTXISR if TXISR or NOTXISR is not specified. TXISR option can only be used when using hardware UART.

**NOTXISR -** If TRANSMIT_BUFFER is greater then zero specifies to not use TBE interrupt for transmitting data. Default is NOTXISR if TXISR or NOTXISR is not specified and XMIT_BUFFER is greater then zero.

**flow control options:**

**RTS=PIN_xx -** Pin to use for RTS flow control. When using FLOW_CONTROL_MODE this pin is driven to the active level when it is ready to receive more data. In SIMPLEX_MODE the pin is driven to the

active level when it has data to transmit. FLOW_CONTROL_MODE can only be use when using RECEIVE_BUFFER.

**RTS_LEVEL=x -** Specifies the active level of the RTS pin, HIGH is active high and LOW is active low. Defaults to LOW if not specified.

**CTS=PIN_xx -** Pin to use for CTS flow control. In both FLOW_CONTROL_MODE and SIMPLEX_MODE this pin is sampled to see if it clear to send data. If pin is at active level and there is data to send it will send next data byte.

**CTS_LEVEL=x -** Specifies the active level of the CTS pin, HIGH is active high and LOW is active low. Default to LOW if not specified.

**FLOW_CONTROL_MODE -** Specifies how the RTS pin is used. For FLOW_CONTROL_MODE the RTS pin is driven to the active level when ready to receive data. Defaults to FLOW_CONTROL_MODE when neither FLOW_CONTROL_MODE or SIMPLEX_MODE is specified. If RTS pin is not specified then this option is not used.

**SIMPLEX_MODE -** Specifies how the RTS pin is used. For SIMPLEX_MODE the RTS pin is driven to the active level when it has data to send. Defaults to FLOW_CONTROL_MODE when neither FLOW_CONTROL_MODE or SIMPLEX_MODE is specified. If RTS pin is not specified then this option is not used.

**Description:**
This directive tells the compiler the baud rate and pins used for serial I/O. This directive takes effect until another RS232 directive is encountered. The #USE DELAY directive must appear before this directive can be used. This directive enables use of built-in functions such as GETC, PUTC, and PRINTF. The functions created with this directive are exported when using multiple compilation units. To access the correct function use the stream identifier.

When using parts with built-in SCI (*[PCD]* UART) and the SCI (*[PCD]* UART) pins are specified, the SCI will be used. If a baud rate cannot be achieved within 3% of the desired value using the current clock rate, an error will be generated. The definition of the RS232_ERRORS is as follows:

No UART:

- Bit 7 is 9th bit for 9 bit data mode (get and put).
- Bit 6 set to one indicates a put failed in float high mode.

With a UART:

- Used only by get:
- Copy of RCSTA register except:
- Bit 0 is used to indicate a parity error.

Definition of the RS232_BUFFER_ERRORS variable is as follows:

- Bit 0 UART Receive overrun error occurred.
- Bit 1 Receive Buffer overflowed.
- Bit 2 Transmit Buffer overflowed.

**Warning:** The device UART will shut down on overflow (3 characters received by the hardware with a GETC() call). The "ERRORS" option prevents the shutdown by detecting the condition and resetting the UART.

**Example Files:**
ex_cust.c

**Examples:**

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

**See Also:**
getc(), putc(), printf(), setup_uart( ), RS2332 I/O overview, kbhit(), puts(), putc_send(), rcv_buffer_bytes(), tx_buffer_bytes(), rcv_buffer_full(), tx_buffer_full(), tx_buffer_available()

## use rtos

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

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

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

**Elements:**
**option** - may be any of the following separated by a comma:

**timer=X** - Where **x** is 0-4 specifying the timer used by the RTOS.

**minor_cycle=time -** Where time is a number followed by s, ms, us, ns. This is the longest time any task will run. Each task's execution rate must be a multiple of this time. The compiler can calculate this if it is not specified.

**statistics -** Maintain min, max, and total time used by each task.

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

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

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

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

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

**See Also:**
#TASK

## #use spi

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

**Elements:**
**option** - may be any of the following separated by a comma:

**MASTER** - Set the device as the master. (default).

**SLAVE -** Set the device as the slave.

**BAUD=n -** Target bits per second, default is as fast as possible.

**CLOCK_HIGH=n -** High time of clock in us (not needed if BAUD= is used). (default=0).

**CLOCK_LOW=n -** Low time of clock in us (not needed if BAUD= is used). (default=0).

**DI=pin -** Optional pin for incoming data.

**DO=pin -** Optional pin for outgoing data.

**CLK=pin -** Clock pin.

**MODE=n -** The mode to put the SPI bus.

**ENABLE=pin -** Optional pin to be active during data transfer.

**LOAD=pin -** Optional pin to be pulsed active after data is transferred.

**DIAGNOSTIC=pin -** Optional pin to the set high when data is sampled.

**SAMPLE_RISE -** Sample on rising edge.

**SAMPLE_FALL -** Sample on falling edge (default).

**BITS=n -** Max number of bits in a transfer. (default=32)

**SAMPLE_COUNT=n -** Number of samples to take (uses majority vote). (default=1

**LOAD_ACTIVE=n -** Active state for LOAD pin (0, 1).

**ENABLE_ACTIVE=n -** Active state for ENABLE pin (0, 1). (default=0)

**IDLE=n -** Inactive state for CLK pin (0, 1). (default=0)

**ENABLE_DELAY=n -** Time in us to delay after ENABLE is activated. (default=0)

**DATA_HOLD=n -** Time between data change and clock change.

**LSB_FIRST -** LSB is sent first.

**MSB_FIRST -** MSB is sent first. (default)

**STREAM=id -** Specify a stream name for this protocol.

**SPI1 -** Use the hardware pins for SPI Port 1.

**SPI2 -** Use the hardware pins for SPI Port 2.

*[PCD]* **SPI3 -** Use the hardware pins for SPI Port 3

*[PCD]* **SPI4 -** Use the hardware pins for SPI Port 4

**FORCE_SW -** Use a software implementation even when hardware pins are specified.

**FORCE_HW -** Use the pic hardware SPI.

**NOINIT -** Do not initialize the hardware SPI Port.

*[PCD]* **XFER16 -** Use 16-bit transfers instead of two 8-bit transfers.

**Description:**
The SPI library contains functions to implement an SPI bus. After setting all of the proper parameters in #USE SPI, the spi_xfer() function can be used to both transfer and receive data on the SPI bus.

The SPI1 and SPI2 options will use the SPI hardware onboard the PIC. The most common pins present on hardware SPI are: DI, DO, and CLK. These pins don't need to

be assigned values through the options; the compiler will automatically assign hardware-specific values to these pins. Consult your PIC's data sheet as to where the pins for hardware SPI are. If hardware SPI is not used, then software SPI will be used. Software SPI is much slower than hardware SPI, but software SPI can use any pins to transfer and receive data other than just the pins tied to the PIC's hardware SPI pins.

The MODE option is more or less a quick way to specify how the stream is going to sample data. MODE=0 sets IDLE=0 and SAMPLE_RISE. MODE=1 sets IDLE=0 and SAMPLE_FALL. MODE=2 sets IDLE=1 and SAMPLE_FALL. MODE=3 sets IDLE=1 and SAMPLE_RISE. There are only these 4 MODEs.

SPI cannot use the same pins for DI and DO. If needed, specify two streams: one to send data and another to receive data.

The pins must be specified with DI, DO, CLK or SPIx, all other options are defaulted as indicated above.

**See Also:**
spi_xfer()

## #use standard_io

**Syntax:**
#use standard_io ***(port)***

**Elements:**
**port** - is A, B, C, D, E, F, G, H, J or ALL

**Description:**
This directive affects how the compiler will generate code for input and output instructions that follow. This directive takes effect until another #USE XXX_IO directive is encountered. The standard method of doing I/O will cause the compiler to generate code to make an I/O pin either input or output every time it is used. On the 5X processors this requires one byte of RAM for every port set to standard I/O.

Standard_io is the default I/O method for all ports.

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

**Example Files:**
ex_cust.c

**Examples:**

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

**See Also:**

#USE FAST_IO, #USE FIXED_IO, General Purpose I/O

## #use timer

**Syntax:**

#use timer **(options)**

**Elements:**

**TIMER=x -** Sets the timer to use as the tick timer. x is a valid timer that the PIC has. Default value is 1 for Timer 1.

**TICK=xx -** Sets the desired time for 1 tick. xx can be used with ns(nanoseconds), us (microseconds), ms (milliseconds), or s (seconds). If the desired tick time can't be achieved it will set the time to closest achievable time and will generate a warning specifying the exact tick time. The default value is 1us.

**BITS=x -** Sets the variable size used by the get_ticks() and set_ticks() functions for returning and setting the tick time. x can be 8 for 8 bits, 16 for 16 bits or 32 for 32bits. The default is 32 for 32 bits.

*[PCD]* **BITS=x -** Sets the variable size used by the get_ticks() and set_ticks() functions for returning and setting the tick time. x can be 8 for 8 bits, 16 for 16 bits, 32 for 32bits or 64 for 64 bits. The default is 32 for 32 bits.

**ISR -** Uses the timer's interrupt to increment the upper bits of the tick timer. This mode requires the the global interrupt be enabled in the main program.

**NOISR -** The get_ticks() function increments the upper bits of the tick timer. This requires that the get_ticks() function be called more often then the timer's overflow rate. NOISR is the default mode of operation.

**STREAM=id -** Associates a stream identifier with the tick timer. The identifier may be used in functions like get_ticks().

**DEFINE=id -** Creates a define named id which specifies the number of ticks that will occur in one second. Default define name if not specified is TICKS_PER_SECOND. Define name must start with an ASCII letter 'A' to 'Z', an ASCII letter 'a' to 'z' or an ASCII underscore ('_').

**COUNTER or COUNTER=x -** Sets up specified timer as a counter instead of timer. x specifies the prescallar to setup counter with, default is1 if x is not specified specified. The function get_ticks() will return the current count and the function set_ticks() can be used to set count to a specific starting value or to clear counter.

**Description:**
This directive creates a tick timer using one of the PIC's timers. The tick timer is initialized to zero at program start. This directive also creates the define TICKS_PER_SECOND as a floating point number, which specifies that number of ticks that will occur in one second.

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

unsigned int16 tick_difference(unsigned int16 current, unsigned int16
previous) {

   return(current - previous);
}

void main(void) {
   unsigned int16 current_tick, previous_tick;
   current_tick = previous_tick = get_ticks();
   while(TRUE) {
      current_tick = get_ticks();
      if(tick_difference(current_tick, previous_tick) > 1000) {
         output_toggle(PIN_B0);
         previous_tick = current_tick;
      }
   }
}
```

**See Also:**
get_ticks(), set_ticks()

## #use touchpad

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

**Elements:**
***RANGE=x -*** *Sets the oscillator charge/discharge current range. If x is L, current is nominally 0.1 microamps. If x is M, current is nominally 1.2 microamps. If x is H, current is nominally 18 microamps. Default value is H (18 microamps).*

***THRESHOLD=x -*** *x is a number between 1-100 and represents the percent reduction in the nominal frequency that will generate a valid key press in software. Default value is 6%.*

***SCANTIME=xxMS -*** *xx is the number of milliseconds used by the microprocessor to scan for one key press. If utilizing multiple touch pads, each pad will use xx milliseconds to scan for one key press. Default is 32ms.*

***PIN=char -*** *If a valid key press is determined on "PIN", the software will return the character "char" in the function touchpad_getc(). (Example: PIN_B0='A')*

**SOURCETIME=xxus -** (CTMU only) **xx** is the number of microseconds each pin is sampled for by ADC during each scan time period. Default is 10us.

**Description:**
This directive will tell the compiler to initialize and activate the Capacitive Sensing Module (CSM)or Charge Time Measurement Unit (CTMU) on the microcontroller. The compiler requires use of the TIMER0 and TIMER1 modules for CSM and Timer1 ADC modules for CTMU, and global interrupts must still be activated in the main program in order for the CSM or CTMU to begin normal operation. For most applications, a higher RANGE, lower THRESHOLD, and higher SCANTIME will result better key press detection. Multiple PIN's may be declared in "options", but they must be valid pins used by the CSM or CTMU. The user may also generate a TIMER0 ISR with TIMER0's interrupt occuring every SCANTIME milliseconds. In this case, the CSM's or CTMU's ISR will be executed first.

**Examples:**
```
#USE TOUCHPAD (THRESHOLD=5, PIN_D5='5', PIN_B0='C')
void main(void){
     char c;
     enable_interrupts(GLOBAL);

     while(1){
         c = TOUCHPAD_GETC();  //will wait until a pin is detected
     }                          //if PIN_B0 is pressed, c will have 'C'
}                               //if PIN_D5 is pressed, c will have '5'
```

**See Also:**
touchpad_state( ), touchpad_getc( ), touchpad_hit( )

## #warning

**Syntax:**
#warning ***text***

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

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

To prevent the warning from being counted as a warning, use this syntax:
***#warning/information text***

**Example Files:**
ex_psp.c

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

**See Also:**
#ERROR

## #word

**Syntax:**
#word **id=x**

**Elements:**
**id -** is a valid C identifier.

**x -** is a C variable or a constant

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

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

**Examples:**
```
#word data = 0x0800

struct  {
  int lowerByte : 8;
  int upperByte : 8;
```

```
} control_word;
#word control_word = 0x85
...
control_word.upperByte = 0x42;

[PCD]
#word data = 0x0860
struct  {
  short C;
  short Z;
  short OV;
  short N;
  short RA;
  short IPL0;
  short IPL1;
  short IPL2;
  int upperByte : 8;
} status_register;
#word status_register = 0x42
...
short zero = status_register.Z;
```

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

## #zero_local_ram

**Syntax:**
#zero_local_ram

**Elements:**
None

**Description:**
This directive causes the compiler to initialize all local variables with no initializer to zero every time the function is invoked. Local variables with an initializer (= after the declaration) are not affected.

**Example Files:**
None

**Examples:**

```
#zero_local_ram
void sample_adc(void {
   int raw_data[1-];           // both raw_data and
```

```
    int sum;                    // sum zero'ed on each call
      . . .
}
```

**See Also:**
#zero_ram, #fill_rom, static

## #zero_ram

**Syntax:**
#zero_ram

**Elements:**
None

**Description:**
This directive zero's out all of the internal registers that may be used to hold variables before program execution begins.

**Example Files:**
ex_cust.c

**Examples:**

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

# BUILT-IN FUNCTIONS

The CCS compiler provides a lot of built-in functions to access and use the PIC microcontroller's peripherals. This makes it very easy for the users to configure and use the peripherals without going into in depth details of the registers associated with the functionality. The functions categorized by the peripherals associated with them are listed on the next page. Click on the function name to get a complete description and parameter and return value descriptions.

## abs( )

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

**Parameters:**
***x*** is a signed 8, 16, or 32 bit int or a float
*[PCD]* **x** is any integer or float type.

**Returns:**
Same type as the parameter.

**Function:**
Computes the absolute value of a number.

**Availability:**
All devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**
```
signed int target,actual;
   ...
error = abs(target-actual);
```

**See Also:**
labs()

# sin( ) cos( ) tan( ) asin( ) acos() atan() sinh() cosh() tanh() atan2()

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

**Parameters:**
***rad*** is a float representing an angle in Radians -2pi to 2pi.
*[PCD]* ***rad*** is any float type representing an angle in Radians -2pi to 2pi.
***val*** is a float with the range -1.0 to 1.0.
*[PCD]* is any float type with the range -1.0 to 1.0.
***Value*** is a float
*[PCD]* ***Value*** is any float type

**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
*[PCD]* rad is a float with a precision equal to **val** representing an angle in Radians -pi/2 to pi/2

*[PCD]* val is a float with a precision equal to **rad** within the range -1.0 to 1.0.

*[PCD]* rad1 is a float with a precision equal to **val** representing an angle in Radians 0 to pi

*[PCD]* rad2 is a float with a precision equal to **val** representing an angle in Radians -pi to pi

*[PCD]* Result is a float with a precision equal to **value**

**Function:**
These functions perform basic Trigonometric functions.

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

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

Domain error occurs in the following cases:

**asin**: when the argument not in the range[-1,+1]
**acos**: when the argument not in the range[-1,+1]
**atan2**: when both arguments are zero

Range error occur in the following cases:

**cosh**: when the argument is too large
**sinh**: when the argument is too large

**Availability:**
All devices

**Requires:**
#INCLUDE <math.h>

**Examples:**

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

**Examples Files:**
ex_tank.c

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

## act_status( )

**Syntax:**
status = act_status();

**Parameters:**
---

**Returns:**
Returns the status of the ACT module. See the device's header file for defines that can be and'ed with result.

**Function:**
Used to get the status of the Active Clock Tuning (ACT) module.

**Availability:**
Devices with an ACT module. See the device's header file for availability.

**Requires:**
---

**Examples:**

```
unsigned int8 Status;
int1 ClockLocked;
Status = act_status();
if((Status & ACT_CLOCK_LOCKED) == 0)
  ClockLoced = FALSE;
else
  ClockLocked = TRUE;
```

**See Also:**
setup_act( )

## adc_done( ) adc2_done( ) adc_done2( )

**Syntax:**
value = adc_done( );
*[PCD]* value = adc_done2( );
*[PCD]* value=adc_done([channel])

**Parameters:**
**adc_done( );** - Nothing required

*[PCD]* **adc_done2( );** - ***channel*** is an optional parameter for specifying the channel to check if the conversion is done. If not specified will use channel specified in the last call to set_adc_channel(), read_adc() or adc_done().

**Returns:**
A short int. TRUE if the A/D converter is done with conversion, FALSE if it is still busy.

**Function:**
Can be polled to determine if the A/D has valid data.

**Availability:**
Only available on devices with built in analog to digital converters
*[PCD]* Only available for dsPIC33EPxxGSxxx family.

**Requires:**
---

**Examples:**

```
int16 value;
setup_adc_ports(sAN0|sAN1, VSS_VDD);
setup_adc(ADC_CLOCK_DIV_4|ADC_TAD_MUL_8);
set_adc_channel(0);
read_adc(ADC_START_ONLY);

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

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

## adc_read( )

**Syntax:**
result=adc_read(register)

**Parameters:**
Register - ADC register to read:

- adc_result
- adc_accumulator
- adc_filter

**Returns:**
int8 or in16 read from the specified register. Return size depends on which register is being read. For example, ADC_RESULT register is 16 bits and ADC_COUNT register is 8-bits.

**Function:**
Reads one of the Analog-to-Digital Converter with Computation (ADC2) Module registers.
**Availability:**
All devices with an ADC2 Module

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

**Examples:**
```
  FilteredResult=adc_read(ADC_FILTER);
```

**See Also:**
ADC Overview, setup_adc(), setup_adc_ports(), set_adc_channel(), read_adc(), #DEVICE, adc_write(), adc_status(), set_adc_trigger()

## adc_status()

**Syntax:**
status=adc_status()

**Parameters:**
Nothing required

**Returns:**
int8 value of the ADSTAT register

**Function:**
Read the current value of the ADSTAT register of the Analog-to-Digital Converter with Computation (ADC2) Module.

**Availability:**
All devices with an ADC2 Module

**Requires:**
---

**Examples:**
```
while((adc_status() & ADC_UPDATING)==0);

Average=adc_read(ADC_FILTER);
```

**See Also:**
ADC Overview, setup_adc(), setup_adc_ports(), set_adc_channel(), read_adc(), #DEVICE, adc_read(), adc_write(), set_adc_trigger()

## adc_write()

**Syntax:**
adc_write(**register, value**)

**Parameters:**
**register** - ADC register to write:

- ADC_REPEAT
- ADC_SET_POINT
- ADC_LOWER_THRESHOLD
- ADC_UPPER_THRESHOLD

**Returns:**
Undefined

**Function:**
Write one of the Analog-to-Digital Converter with Computation (ADC2) Module registers.

**Availability:**
All devices with an ADC2 Module

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

**Examples:**
```
adc_write(ADC_SET_POINT, 300);
```

**See Also:**
ADC Overview, setup_adc(), setup_adc_ports(), set_adc_channel(), read_adc(), #DEVICE, adc_read(), adc_status(), set_adc_trigger()

## assert( )

**Syntax:**
assert (***condition***);

**Parameters:**
***condition*** is any relational expression

**Returns:**
---

**Function:**
This function tests the condition and if FALSE will generate an error message on STDERR (by default the first USE RS232 in the program). The error message will include the file and line of the assert(). No code is generated for the assert() if you #define NODEBUG. In this way you may include asserts in your code for testing and quickly eliminate them from the final program.

**Availability:**
All Devices

**Requires:**
assert.h and #USE RS232

**Examples:**
```
  assert( number_of_entries<TABLE_SIZE );

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

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

## atoe( )

**Syntax:**
atoe(**string**);

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

**Returns:**
Result is a floating point number

**Function:**
Converts the string passed to the function into a floating point representation. If the result cannot be represented, the behavior is undefined. This function also handles E format numbers.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**
```
char string [10];
float32 x;

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

**See Also:**
atoi(), atol(), atoi32(), atof(), printf()

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

**Syntax:**
result = atof (***string***)
*[PCD]* or
result = atof48(***string***)
or
result=atof64(***string***)
or
result-strtof48(***string***))

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

**Returns:**
Result is a floating point number
*[PCD]* Result is a floating point number in single, extended or double precision format

**Function:**
Converts the string passed to the function into a floating point representation. If the result cannot be represented, the behavior is undefined.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**
```
char string [10];
float x;

strcpy (string, "123.456");
x = atof(string);
// x is now 123.456
```

**Example Files:**
ex_tank.c

**See Also:**
atoi(), atol(), atoi32(), printf()

## atoi( ) atol( ) atoi32( ) atol32( ) atoi48( ) atoi64( )

**Syntax:**
ivalue = atoi(*string*)
ivalue = atol(*string*)
i32value = atoi32(*string*)
i32value = atol32(string)
*[PCD]* i48value = atoi48(string);
*[PCD]* i64value = atoi64(string);

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

**Returns:**
ivalue is an 8 bit int
ivalue is a 16 bit int
i32value is a 32 bit int
*[PCD]* i48value is a 48 bit int
*[PCD]* i64value is a 64 bit int

**Function:**
Converts the string passed to the function into an ***int*** representation. Accepts both decimal and hexadecimal argument. If the result cannot be represented, the behavior is undefined.

**Availability:**
All devices

**Requires:**
#INCLUDE<stdlib.h>

**Examples:**
```
char string[10];
int x;

strcpy(string,"123");
x = atoi(string);     // x is now 123
```

**Example Files:**
input.c

**See Also:**
printf( )

## at_clear_interrupts( )

**Syntax:**
at_clear_interrupts(**interrupts**);

**Parameters:**
**interrupts** - an 8-bit constant specifying which AT interrupts to disable. The constants are defined in the device's header file as:
- AT_PHASE_INTERRUPT
- AT_MISSING_PULSE_INTERRUPT
- AT_PERIOD_INTERRUPT
- AT_CC3_INTERRUPT
- AT_CC2_INTERRUPT
- AT_CC1_INTERRUPT

**Returns:**
---

**Function:**
To disable the Angular Timer interrupt flags. More than one interrupt can be cleared at a time by or'ing multiple constants together in a single call, or calling function multiple times for each interrupt to clear.

**Availability:**
All Devices with an AT module

**Requires:**
Constants defined in the device's header file

**Examples:**
```
#INT-AT1
void1_isr(void){
   if(at_interrupt_active(AT_PERIOD_INTERRUPT))
   {
      handle_period_interrupt();
      at_clear_interrupts(AT_PERIOD_INTERRUPT);
   }
   if(at_interrupt(active(AT_PHASE_INTERRUPT);
   {
      handle_phase_interrupt();
      at_clear_interrupts(AT_PHASE_INTERRUPT);
   }
}
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_disable_interrupts( )

**Syntax:**
at_disable_interrupts(**interrupts**);

**Parameters:**
**interrupts** - an 8-bit constant specifying which AT interrupts to disable. The constants are defined in the device's header file as:
- AT_PHASE_INTERRUPT
- AT_MISSING_PULSE_INTERRUPT
- AT_PERIOD_INTERRUPT
- AT_CC3_INTERRUPT
- AT_CC2_INTERRUPT
- AT_CC1_INTERRUPT

**Returns:**
---

**Function:**
To disable the Angular Timer interrupts. More than one interrupt can be disabled at a time by or'ing multiple constants together in a single call, or calling function multiple times for eadch interrupt to be disabled.

**Availability:**
All Devices with an AT module

**Requires:**
Constants defined in the device's header file

**Examples:**
```
at_disable_interrupts(AT_PHASE_INTERRUPT);
at_disable_interrupts(AT_PERIOD_INTERRUPT|AT_CC1_INTERRUPT);
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_enable_interrupts( )

**Syntax:**
at_enable_interrupts(**interrupts**);

**Parameters:**
**interrupts** - an 8-bit constant specifying which AT interrupts to enable. The constants are defined in the device's header file as:
- AT_PHASE_INTERRUPT
- AT_MISSING_PULSE_INTERRUPT
- AT_PERIOD_INTERRUPT
- AT_CC3_INTERRUPT
- AT_CC2_INTERRUPT
- AT_CC1_INTERRUPT

**Returns:**
---

**Function:**
To enable the Angular Timer interrupts. More than one interrupt can be enabled at a time by or'ing multiple constants together in a single call, or calling function multiple times for each interrupt to be enabled.

**Availability:**
All Devices with an AT module

**Requires:**
Constants defined in the device's header file

**Examples:**
```
at_enable_interrupts(AT_PHASE_INTERRUPT);
at_enable_interrupts(AT_PERIOD_INTERRUPT|AT_CC1_INTERRUPT);
```

**See Also:**
`setup_at(), at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point(), at_get_set_point_error(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status()`

## at_get_capture( )

**Syntax:**
result=at_get_capture(**which**);;

**Parameters:**
which - an 8-bit constant specifying which AT Capture/Compare module to get the capture time from, can be 1, 2 or 3.

**Returns:**
A 16-bit integer

**Function:**
To get one of the Angular Timer Capture/Compare modules capture time.

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
result1=at_get_capture(1);
result2=at_get_capture(2);
```

**See Also:**
setup_at(), at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_status()

## at_get_missing_pulse_delay( )

**Syntax:**
result=at_get_missing_pulse_delay();

**Parameters:**
---

**Returns:**
A 16-bit integer

**Function:**
To setup the Angular Timer Missing Pulse Delay

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
result=at_get_missing_pulse_delay();
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_get_period( )

**Syntax:**
result=at_get_period();

**Parameters:**
---

**Returns:**
A 16-bit integer. The MSB of the returned value specifies whether the period counter rolled over one or more times. 1 - counter rolled over at least once, 0 - value returned is valid.

**Function:**
To get one of the Angular Timer Measure Period.

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
result=at_get_period();
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_get_phase_counter( )

**Syntax:**
result=at_get_phase_counter();

**Parameters:**
---

**Returns:**
A 16-bit integer.

**Function:**
To get one of the Angular Timer Phase Counter.

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
result=at_get_phase_counter();
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_get_resolution( )

**Syntax:**
result=at_get_resolution();

**Parameters:**
---

**Returns:**
A 16-bit integer.

**Function:**
To get one of the Angular Timer Resolution.

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
result=at_get_resolution();
```

**See Also:**
at_set_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_get_set_point( )

**Syntax:**
result=at_get_set_point();

**Parameters:**
---

**Returns:**
A 16-bit integer.

**Function:**
To get one of the Angular Timer Set Point.

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
result=at_get_set_point();
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_get_set_point_error( )

**Syntax:**
result=at_get_set_point_error();

**Parameters:**
---

**Returns:**
A 16-bit integer.

**Function:**
To get one of the Angular Timer Set Point Error, the error of the measured period value compared to the threshold setting.

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
result=at_get_set_point_error();
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_get_status( )

**Syntax:**
result=at_get_status();

**Parameters:**
---

**Returns:**
An 8-bit integer. The possible results are defined in the device's header file as:
- AT_STATUS_PERIOD_AND_PHASE_VALID
- AT_STATUS_PERIOD_LESS_THEN_PREVIOUS

**Function:**
To get one of the Angular Timer module.

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
if((at_get_status()&AT_STATUS_PERIOD_AND_PHASE_VALID)==
AT_STATUS_PERIOD_AND_PHASE_VALID
{
   Period=at_get_period();
   Phase=at_get_phase();
}
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), setup_at()

## at_interrupt_active( )

**Syntax:**
result=at_interrupt_active(**interrupt**);

**Parameters:**
**interrupts** - an 8-bit constant specifying which AT interrupts to check if its flag is set. The constants are defined in the device's header file as:
- AT_PHASE_INTERRUPT
- AT_MISSING_PULSE_INTERRUPT
- AT_PERIOD_INTERRUPT
- AT_CC3_INTERRUPT
- AT_CC2_INTERRUPT
- AT_CC1_INTERRUPT

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

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

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
#INT-AT1
void1_isr(void)
{
   if(at_interrupt_active(AT_PERIOD_INTERRUPT))
   {
      handle_period_interrupt();
      at_clear_interrupts(AT_PERIOD_INTERRUPT);
   }
   if(at_interrupt(active(AT_PHASE_INTERRUPT);
   {
      handle_phase_interrupt();
      at_clear_interrupts(AT_PHASE_INTERRUPT);
   }
}
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_set_compare_time( )

**Syntax:**
at_set_compare_time(**which, compare_time**);

**Parameters:**
**which** - an 8-bit constant specifying which AT Capture/Compare module to set the compare time for, can be 1, 2, or 3.

**compare_time** - a 16-bit constant or variable specifying the value to trigger an interrupt/ouput pulse.

**Returns:**
---

**Function:**
To set one of the Angular Timer Capture/Compare module's compare time.

**Availability:**
All Devices with an AT module

**Requires:**
Constants defined in the device's header file

**Examples:**
```
at_set_compare_time(1,0x1FF);
at_set_compare_time(3,compare_time);}
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_get_capture(), at_get_status(), setup_at()

## at_set_missing_pulse_delay( )

**Syntax:**
at_set_missing_pulse_delay(pulse_delay);

**Parameters:**
**pulse_delay** - a signed 16-bit constant or variable to set the missing pulse delay.

**Returns:**
---

**Function:**
To setup the Angular Timer Missing Pulse Delay

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
at_set_missing_pulse_delay(pulse_delay);
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_set_resolution( )

**Syntax:**
at_set_resolution(**resolution**);

**Parameters:**
**resolution** - a 16-bit constant or variable to set the resolution.

**Returns:**
---

**Function:**
To setup the Angular Timer Resolution

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
at_set_resolution(resolution);
```

**See Also:**
at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## at_set_set_point( )

**Syntax:**
at_set_set_point(**set_point**);

**Parameters:**
**resolution** - a 16-bit constant or variable to set the resolution.

**Returns:**
---

**Function:**
To setup the Angular Timer Set Point

**Availability:**
All Devices with an AT module

**Requires:**
---

**Examples:**
```
at_set_set_point(set_point);
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()
.

## at_setup_cc( )

**Syntax:**
at_setup_cc(**which, settings**);

**Parameters:**
**which** - an 8-bit constant specifying which AT Capture/Compare to setup, can be 1, 2 or 3.

**settings** - a 16-bit constant specifying how to setup the specified AT Capture/Compare module. See the device's header file for all options. Some of the typical options include:
- AT_CC_ENABLED
- AT_CC_DISABLED
- AT_CC_CAPTURE_MODE
- AT_CC_COMPARE_MODE
- AT_CAPTURE_FALLING_EDGE
- AT_CAPTURE_RISING_EDGE

**Returns:**
---

**Function:**
To setup one of the Angular Timer Capture/Compare modules to the specified settings.

**Availability:**
All Devices with an AT module

**Requires:**
Constants defined in the device's header file

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

at_setup_cc(2,AT_CC_ENABLED|AT_CC_CAPTURE_MODE|
AT_CC_ACTIVE_HIGH);
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_set_compare_time(), at_get_capture(), at_get_status(), setup_at()

## bit_clear( )

**Syntax:**
bit_clear(***var***, ***bit***)

**Parameters:**
***var*** may be a any bit variable (any lvalue)
***bit*** is a number 0- 31 63 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 similar to: var &= ~(1<<bit);

**Availability:**
All Devices

**Requires:**
---

**Examples:**
```
int x;
x=5;
bit_clear(x,2);       // x is now 1
```

**Example Files:**
ex_patg.c

**See Also:**
bit_set(), bit_test()

## bit_first( )

**Syntax:**
N = bit_first (***value, var***)

**Parameters:**
***value*** is a 0 to 1 to be shifted in
***var*** is a 16 bit integer

**Returns:**
An 8-bit integer

**Function:**
This function sets N to the 0 based position of the first occurrence of value. The search starts from the right or least significant bit.

**Availability:**
24-bit Devices (PIC24, 30F/33F)

**Requires:**
---

**Examples:**
```
int16 var = 0x0033;
Int8 N = 0;
// N = 2
N = bit_first (0, var);
```

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

## bit_last( )

**Syntax:**
N = bit_last (***value, var***)
N = bit_last(***var***)

**Parameters:**
***value*** is a 0 to 1 to search for
***var*** is a 16 bit integer

**Returns:**
An 8-bit integer

**Function:**
The first function will find the first occurrence of value in the var starting with the most significant bit.
The second function will note the most significant bit of var and then search for the first different bit.
Both functions return a 0 based result.

**Availability:**
24-bit Devices (PIC24, 30F/33F)

**Requires:**
---

**Examples:**

```
                          //Bit pattern 11101110 11111111
  Int16 var = 0xEEFF;
  Int8 N = 0;             //N is assigned 12
  N = bit_last (0, var);  //N is assigned 12
  N = bit_last(var)
```

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

## bit_set( )

**Syntax:**
bit_set(***var***, ***bit***)

**Parameters:**
***var*** may be any variable (any lvalue)

***bit*** is a number from 0 to the highest bit number in the type, 0 is the least significant bit

**Returns:**
Undefined

**Function:**
Sets the specified bit in the given variable. The least significant bit is 0.
This function is the similar to: var |= (1<<bit);
For example, for a 16-bit variable, the bit number may be 0-15,

**Availability:**
All Devices

**Requires:**
---

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

**Example Files:**
ex_patg.c

**See Also:**
bit_clear(), bit_test()

## bit_test( )

**Syntax:**
value = bit_test (***var***, ***bit***)

**Parameters:**
***var*** may be any variable (any lvalue)
***bit*** is a number from 0 to the highest bit number in the type, 0 is the least significant bit

**Returns:**
0 or 1

**Function:**
Tests the specified bit in the given variable. The least significant bit is 0.
This function is more efficient than, but otherwise similar to ((var & (1<<bit)) != 0)
For example, for a 16-bit variable, the bit number may be 0-15,

**Availability:**
All Devices

**Requires:**
---

**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

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

## brownout_enable( )

**Syntax:**
brownout_enable (***value***)

**Parameters:**
***value – TRUE or FALSE***

**Returns:**
Undefined

**Function:**
Enable or disable the software controlled brownout. Brownout will cause the PIC to reset if the power voltage goes below a specific set-point.

**Availability:**
This function is only available on devices with a software controlled brownout. This may also require a specific configuration bit/fuse to be set for the brownout to be software controlled.

**Requires:**
---

**Examples:**
```
brownout_enable(TRUE);
```

**See Also:**
restart_cause()

## bsearch( )

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

**Parameters:**
***key -*** Object to search for
***base -*** Pointer to array of search data
***num -*** Number of elements in search data
***width -*** Width of elements in search data
***compare -*** Function that compares two elements in search data

**Returns:**
**bsearch** returns a pointer to an occurrence of key in the array pointed to by base. If key is not found, the function returns NULL. If the array is not in order or contains duplicate records with identical keys, the result is unpredictable.

**Function:**
Performs a binary search of a sorted array.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**
```
int nums[5]={1,2,3,4,5};
int compar(const void *arg1,const void *arg2);

void main() {
   int *ip, key;
   key = 3;
   ip = bsearch(&key, nums, 5, sizeof(int), compar);
```

```
}
int compar(const void *arg1,const void *arg2)  {
   if ( * (int *) arg1 < ( * (int *) arg2) return -1
   else if ( * (int *) arg1 == ( * (int *) arg2) return 0
   else return 1;
}
```

**See Also:**
qsort()

## calloc( )

**Syntax:**
ptr=calloc(***nmem***, ***size***)

**Parameters:**
***nmem*** is an integer representing the number of member objects
**size** is the number of bytes to be allocated for each one of them.

**Returns:**
A pointer to the allocated memory, if any. Returns null otherwise.

**Function:**
The **calloc** function allocates space for an array of nmem objects whose size is specified by size.
The space is initialized to all bits zero.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**

```
int * iptr;
iptr=calloc(5,10);   // iptr will point to a block of memory of
                     // 50 bytes all initialized to 0
```

**See Also:**
realloc(), free(), malloc()

## ceil( )

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

**Parameters:**
***value*** is a float
*[PCD]* ***value*** is any float type

**Returns:**
A float
*[PCD]* A float with precision equal to ***value***

**Function:**
Computes the smallest integer value greater than the argument. CEIL(12,67) is 13,00.

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

**Examples:**

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

**See Also:**
floor()

## clc1_setup_gate( ) clc2_setup_gate( ) clc3_setup_gate( ) clc4_setup_gate( )

**Syntax:**
clc1_setup_gate(gate, mode);
clc2_setup_gate(gate, mode);
clc3_setup_gate(gate, mode);
clc4_setup_gate(gate, mode);

**Parameters:**
**gate** – selects which data gate of the Configurable Logic Cell (CLC) module to setup, value can be 1 to 4.

**mode** – the mode to setup the specified data gate of the CLC module into. The options are:

clc_gate_and
clc_gate_nand
clc_gate_nor
clc_gate_or
clc_gate_clear
clc_gate_set

**Returns:**
Undefined
[PCD] Undefined with precision equal to ***value***

**Function:**
Sets the logic function performed on the inputs for the specified data gate.

**Availability:**
Devices with a CLC module

**Requires:**
Undefined

**Examples:**
```
clc1_setup_gate(1, CLC_GATE_AND);
clc1_setup_gate(2, CLC_GATE_NAND);
clc1_setup_gate(3, CLC_GATE_CLEAR);
clc1_setup_gate(4, CLC_GATE_SET);
```

**See Also:**
setup_clcx(), clcx_setup_input()

## clc1_setup_input() clc2_setup_input() clc3_setup_input() clc4_setup_input()

**Syntax:**
clc1_setup_input(**input**, **selection**);
clc2_setup_input**(input**, **selection**);
clc3_setup_input(**input**, **selection**);
clc4_setup_input(**input**, **selection**);

**Parameters:**
**input** – selects which input of the Configurable Logic Cell (CLC) module to setup, value can be 1 to 4.

**selection** – the actual input for the specified input that is actually connected to the data gates of the CLC module. The options are:

clc_input_0
clc_input_1
clc_input_2
clc_input_3
clc_input_4
clc_input_5
clc_input_6
clc_input_7

**Returns:**
Undefined

**Function:**
Sets the input for the specified input number that is actually connected to all four data gates of the CLC module. Please refer to the table CLCx DATA INPUT SELECTION in the device's datasheet to determine which of the above selections corresponds to actual input pin or peripheral of the device.

**Availability:**
Devices with a CLC module

**Requires:**
Undefined

**Examples:**
```
clc1_setup_input(1, CLC_INPUT_0);
clc1_setup_input(2, CLC_INPUT_1);
clc1_setup_input(3, CLC_INPUT_2);
clc1_setup_input(4, CLC_INPUT_3);
```

**See Also:**
setup_clcx(), clcx_setup_gate()

## clear_dmt( )

**Syntax:**
clear_dmt();

**Parameters:**
---

**Returns:**
---

**Function:**
Used to clear the Deadman Timer (DMT) peripheral.

**Availability:**
Only on devices that have the DMT peripheral.

**Requires:**
---

**Examples:**
```
  if((dmt_status() & DMT_CLEAR_WINDOW_OPEN) == DMT_CLEAR_WINDOW_OPEN)
       clear_dmt();
```

**See Also:**
read_dmt(), disable_dmt(), enable_dmt(), dmt_status(), setup_dmt()

## clear_interrupt( )

**Syntax:**
clear_interrupt(***level***)

**Parameters:**
**level** - a constant defined in the devices.h file

**Returns:**
Undefined

**Function:**
Clears the interrupt flag for the given level. This function is designed for use with a specific interrupt, thus eliminating the GLOBAL level as a possible parameter. Some chips that have interrupt on change for individual pins allow the pin to be specified like INT_RA1.

**Availability:**
All Devices

**Requires:**
---

**Examples:**
```
  clear_interrupt(int_timer1);
```

**See Also:**
enable_interrupts() , enable_interrupts , #INT , #INT , Interrupts Overview
disable_interrupts(), interrupt_actvie()

## clear_pwm1_interrupt( ) clear_pwm2_interrupt( ) clear_pwm3_interrupt( ) clear_pwm4_interrupt( ) clear_pwm5_interrupt( ) clear_pwm6_interrupt( )

**Syntax:**
clear_pwm1_interrupt (***interrupt***)
clear_pwm2_interrupt (***interrupt***)
clear_pwm3_interrupt (***interrupt***)
clear_pwm4_interrupt (***interrupt***)
clear_pwm5_interrupt (***interrupt***)
clear_pwm6_interrupt (***interrupt***)

**Parameters:**
***interrupt*** - 8-bit constant or variable. Constants are defined in the device's header file as:

pwm_period_interrupt
pwm_duty_interrupt
pwm_phase_interrupt
pwm_offset_interrupt

**Returns:**
Undefined

**Function:**
Clears one of the above PWM interrupts, multiple interrupts can be cleared by or'ing multiple options together.

**Availability:**
Devices with a 16-bit PWM module

**Requires:**
---

**Examples:**
```
clear_pwm1_interrupt(PWM_PERIOD_INTERRUPT);
clear_pwm1_interrupt(PWM_PERIOD_INTERRUPT | PWM_DUTY_INTERRUPT)
```

**See Also:**
setup_pwm(), set_pwm_duty(), set_pwm_phase(), set_pwm_period(), set_pwm_offset(), enable_pwm_interrupt(), disable_pwm_interrupt(), pwm_interrupt_active()

## cog_restart( ) cog2_restart( ) cog3_restart( ) cog4_restart()

**Syntax:**
cog_restart();
cog2_restart();
cog3_restart();
cog4_restart();

**Parameters:**
---

**Returns:**
---

**Function:**
To restart the Complementary Output Generator (COG) module after an auto-shutdown event occurs, when not using auto-restart option of module.

**Availability:**
Devices with a COG module

**Requires:**
---

**Examples:**
```
if(cog_status()==COG_AUTO_SHUTDOWN)
  cog_restart();
```

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

## cog_status( ) cog2_status( ) cog3_status( ) cog4_status( )

**Syntax:**
value=cog_status();
value=cog2_status();
value=cog3_status();
value=cog4_status();

**Parameters:**
---

**Returns:**
value - the status of the COG module

**Function:**
To determine if a shutdown event occurred on the Complementary Output Generator (COG) module.

**Availability:**
Devices with a 16-bit PWM module

**Requires:**
---

**Examples:**
```
  if(cog_status()==COG_AUTO_SHUTDOWN)
    cog_restart();
```

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

## crc_calc(mode )

**Syntax:**
Result = crc_calc (**data,[width]**);
Result = crc_calc(**ptr,len,[width]**);
Result = crc_calc8(**data,[width]**);
Result = crc_calc8(**ptr,len,[width]**);
Result = crc_calc16(**data,[width]**); //same as crc_calc( )
Result = crc_calc16(**ptr,len,[width]**); //same as crc_calc( )
*[PCD]* Result = crc_calc32(**data,[width]**);
*[PCD]* Result = crc_calc32(**ptr,len,[width]**);

**Parameters:**
**data**- This is one double word, word or byte that needs to be processed when using
crc_calc16( )
crc_calc8( )
*[PCD]* crc_calc32( )

**ptr**- is a pointer to one or more double words, words or bytes of data

**len**- number of double words, words or bytes to process for function calls
crc_calc16( )
crc_calc8( )
*[PCD]* crc_calc32( )

**width**- optional parameter used to specify the input data bit width to use with the functions
crc_calc16( )
crc_calc8( )
*[PCD]* crc_calc32( )

If not specified, it defaults to the width of the return value of the function
8-bit for crc_calc8( )
16-bit for crc_calc16( )
*[PCD]* 32-bit for crc_calc32( )

**Returns:**
Returns the result of the final CRC calculation.

**Function:**
Calculates the CRC of the passed data using the CRC engine. The function that should be used to do the calculation depends on the CRC polynomial used. For polynomials less than or equal 8 bits, crc_calc8() should be used. For polynomials greater than 8 bits, crc_calc16() should be used. Data widths less than or equal to 16 bits are supported.

*[PCD]* Calculates the CRC of the passed data using the CRC engine. The crc_calc32() function is only available for device with a 32 bit CRC peripheral. The function that should be used to do the calculation depends on the CRC polynomial used. For polynomials less than or equal to 8 bits, crc_calc8() should be used. For polynomials greater than 8 bits and less than or equal to 16 bits, crc_calc16() should be used. For polynomials greater than 16 bits, crc_calc32() should be used. For devices with a 32 bit CRC peripheral, data widths less than or equal to 32 bits are supported, and for device with a 16 bit CRC peripheral data widths less than or equal to 16 bits are supported.

**Availability:**
Only Devices with a built-in CRC module

**Requires:**
---

**Examples:**
```
int16 data[8];
Result = crc_calc(data,8);
```

**Example Files:**
ex_crc_hw.c

**See Also:**
setup_crc(); crc_init()

## crc_init(mode)

**Syntax:**
crc_init (***data***);

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

**Returns:**
Undefined

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

**Availability:**
Only Devices with a built-in CRC module

**Requires:**
---

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

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

**See Also:**
setup_crc(), crc_calc(), crc_calc8()

## crc_read( )

**Syntax:**
value = read();

**Parameters:**
---

**Returns:**
A 16-bit integer.

**Function:**
Returns the current CRC Accumulator value.

**Availability:**
On devices with a Cyclic Redundancy Check (CRC) module.

**Requires:**
---

**Examples:**

```
int16 value;
value = crc_read();
```

**See Also:**
setup_crc( ), crc_init( ), crc_calc( ), crc_write( )

## crc_write( )

**Syntax:**
crc_write(***data, [data_width]***);

**Parameters:**

***data*** is the 16 bit value to write

***data_width*** is an optional parameter used to specify the width of the input data.

**Returns:**
Undefined

**Function:**
Used to write data into the CRC data registers.

**Availability:**
On devices with a Cyclic Redundancy Check (CRC) module.

**Requires:**
---

**Examples:**

```
crc_write(data);
```

**See Also:**
setup_crc( ), crc_init( ), crc_calc( ), crc_read( )

## cwg_restart( ) cwg2_restart( ) cwg3_restart( )

**Syntax:**
cwg_restart( );
cwg2_restart( );
cwg3_restart( );

**Parameters:**
---

**Returns:**
---

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

**Availability:**
Devices with a CWG module

**Requires:**
---

**Examples:**

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

**See Also:**
setup_cwg( ), cwg_status( )

## cwg_status( ) cwg2_status( ) cwg3_status( )

**Syntax:**
value = cwg_status( );
value = cwg2_status( );

value = cwg3_status( );

**Parameters:**
---

**Returns:**
The status of the CWG module

**Function:**
To determine if a shutdown event occurred causing the module to auto-shutdown.

**Availability:**
Devices with a CWG module

**Requires:**
---

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

**See Also:**
setup_cwg( ), cwg_restart( )

## dac_write( )

**Syntax:**
dac_write (**value**);
dac_write2 (**value**);
dac_write3(**value**);
dac_write4(**value**);
dac_write5(**value**);
dac_write6(**value**);
dac_write7(**value**);
dac_write8(**value**);
*[PCD]* dac_write (**channel, value**);
*[PCD]* dac_write (**module, value**);
*[PCD]* dac_write (**module, value, [low_value]**);

**Parameters:**
***value*** - 8-bit or 16-bit integer value to be written to the DAC module

*[PCD]* ***channel*** - 16-bit integer value to be written to the DAC module channel: Channel to be written to. Constants are:

DAC_RIGHT
DAC_DEFAULT
DAC_LEFT

*[PCD]* ***module*** - DAC module to write value to.

[PCD] ***low_value*** - Optional 16-bit integer value for devices with an Analog Comparator with Slope Compensation DAC peripheral to set the DAC low data value. In Hysteric, Slope Generator and Triangle modes, this specifies the low data value and/or limit for the DAC module.

**Returns:**
---

**Function:**
This function will write a 8-bit or 16-bit integer to the specified DAC module.

**Availability:**
Devices with an analog-to-digital converter (DAC).

**Requires:**
---

**Examples:**

```
dac_write(20);

[PCD]
dac_write(DAC_RIGHT, 500);
dac_write(1, DacValue);
dac_write(1, DacValue, DacLowValue);
```

**See Also:**
setup_dac( ), DAC Overview, See header file for device selected

## dci_data_received( )

**Syntax:**
dci_data_received()

**Parameters:**
---

**Returns:**
An int1. Returns true if the DCI module has received data.

**Function:**
Use this function to poll the receive buffers. It acts as a kbhit() function for DCI.

**Availability:**
Devices with a DCI

**Requires:**
---

**Examples:**
```
while(1)
{
       if(dci_data_received())
      {                              //read data, load buffers, etc…
      }
}
```

**See Also:**
DCI Overview, setup_dci( ), dci_start( ), dci_write( ), dci_read( ), dci_transmit_ready( )

## dci_read( )

**Syntax:**
dci_read(***left_ channe****l, **right_ channel**);*

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

*right_channel*- A pointer to a signed int16 that will hold the incoming audio data for the right channel (on a stereo system). This data is received on the bus after the data in *left channel.*

**Returns:**
Undefined

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

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

**Availability:**
Devices with a DCI

**Requires:**
---

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

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

## dci_start( )

**Syntax:**
*dci_start();*

**Parameters:**
---

**Returns:**
Undefined

**Function:**
Starts the DCI module's transmission. DCI operates in a continous transmission mode (unlike other transmission protocols that transmit only when they have data). This function starts the transmission. This function is primarily provided to use DCI in conjunction with DMA

**Availability:**
Devices with a DCI

**Requires:**
---

**Examples:**

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

**See Also:**
DCI Overview, setup_dci( ), dci_write( ), dci_read( ), dci_transmit_ready( ), dci_data_received( )

## dci_transmit_ready( )

**Syntax:**
*dci_transmit_ready()*

**Parameters:**
---

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

**Function:**
Use this function to poll the transmit buffers

**Availability:**
Devices with a DCI

**Requires:**
---

**Examples:**

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

**See Also:**
DCI Overview, setup_dci( ), dci_start( ), dci_write( ), dci_read( ), dci_data_received( )

## dci_write( )

**Syntax:**
dci_write(**left_channel**, **right_channel**);

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

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

**Returns:**
Undefined

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

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

**Availability:**
Devices with a DCI

**Requires:**
---

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

**See Also:**
DCI Overview, setup_dci( ), dci_start( ), dci_read( ), dci_transmit_ready( ), dci_data_received( )

## delay_cycles( )

**Syntax:**
delay_cycles (***count***)

**Parameters:**
***count*** - a constant 1-255

**Returns:**
Undefined

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

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

**Availability:**
All Devices

**Requires:**
---

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

**Example Files:**
ex_cust.c

**See Also:**
delay_us(), delay_ms()

## delay_ms( )

**Syntax:**
delay_ms (***time***)

**Parameters:**
***time*** - a variable 0-65535(int16) or a constant 0-65535

Note: Previous compiler versions ignored the upper byte of an int16, now the upper byte affects the time.

**Returns:**
Undefined

**Function:**
This function will create code to perform a delay of the specified length. Time is specified in milliseconds. This function works by executing a precise number of instructions to cause the requested delay. It does not use any timers. If interrupts are enabled the time spent in an interrupt routine is not counted toward the time.

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

**Availability:**
All Devices

**Requires:**
#USE_DELAY

**Examples:**
```
#use delay (clock=20000000)
delay_ms( 2 );

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

**Example Files:**
ex_sqw.c

**See Also:**
delay_us(), delay_cycles(), #USE DELAY

## delay_us( )

**Syntax:**
delay_us (***time***)

**Parameters:**
***time*** - a variable 0-65535(int16) or a constant 0-65535

Note: Previous compiler versions ignored the upper byte of an int16, now the upper byte affects the time.

**Returns:**
Undefined

**Function:**
Creates code to perform a delay of the specified length. Time is specified in microseconds. Shorter delays will be INLINE code and longer delays and variable delays are calls to a function. This function works by executing a precise number of instructions to cause the requested delay. It does not use any timers. If interrupts are enabled the time spent in an interrupt routine is not counted toward the time.

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

**Availability:**
All Devices

**Requires:**
#USE_DELAY

**Examples:**
```
#use delay(clock=20000000)

do {
output_high(PIN_B0);
delay_us(duty);
output_low(PIN_B0);
delay_us(period-duty);
} while(TRUE);
```

**Example Files:**
ex_sqw.c

**See Also:**
delay_ms(), delay_cycles(), #USE DELAY

## disable_dmt( )

**Syntax:**
disable_dmt();

**Parameters:**
---

**Returns:**
---

**Function:**
Disable the Deadman Timer (DMT). This function only works if the DMT_SW configuration fuses has been set. If the DMT configuration fuse is set, then the DMT is always enabled.

**Availability:**
Only on devices that have the DMT peripheral.

**Requires:**
---

**Examples:**
```
disabled_dmt();
```

**See Also:**
clear_dmt(), read_dmt(), enable_dmt(), dmt_status(), setup_dmt()

## disable_interrupts( )

**Syntax:**
disable_interrupts (***level***)
*[PCD]* disable_interrupts *(**name**)*
*[PCD]* disable_interrupts *(**INTR_XX**)*
*[PCD]* disable_interrupts *(**expression**)*

**Parameters:**
***level*** - a constant defined in the devices .h file

*[PCD]* ***name*** *- a constant defined in the devices .h file*

*[PCD]* **INTR_XX** – Allows user selectable interrupt options like **intr_normal, intr_alternate, intr_level**

*[PCD]* **expression** – A non-constant expression

**Returns:**
Undefined
*[PCD]* When **intr_levelx** is used as a parameter, this function will return the previous level.

**Function:**
Disables the interrupt at the given level. The GLOBAL level will not disable any of the specific interrupts but will prevent any of the specific interrupts, previously enabled to be active. Valid specific levels are the same as are used in #INT_xxx and are listed in the devices .h file. GLOBAL will also disable the peripheral interrupts on devices that have it.

Note that it is not necessary to disable interrupts inside an interrupt service routine since interrupts are automatically disabled. Some chips that have interrupt on change for individual pins allow the pin to be specified like INT_RA1.

*[PCD]*
Disables the interrupt for the given name. Valid specific names are the same as are used in #INT_xxx and are listed in the devices .h file. Note that it is not necessary to disable interrupts inside an interrupt service routine since interrupts are automatically disabled.

**intr_glogal** – Disables all interrupts that can be disabled

**intr_normal** – Use normal vectors for the ISR

**intr_alternate** – Use alternate vectors for the ISR

**intr_level0 / intr_level7** – Disables interrupts at this level and below, enables interrupts above this level

**intr_cn_pin|pin_xx** – Disables a CN pin interrupts

**expression** – Disables interrupts during evaluation of the expression.

**Availability:**
Some Devices (PCM and PCH) with interrupts and all 24-bit (PCD) devices.

**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

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

## disable_pwm1_interrupt( ) disable_pwm2_interrupt( ) disable_pwm3_interrupt( ) disable_pwm4_interrupt( ) disable_pwm5_interrupt( ) disable_pwm6_interrupt( )

**Syntax:**
disable_pwm1_interrupt (***interrupt***)
disable_pwm2_interrupt (***interrupt***)
disable_pwm3_interrupt (***interrupt***)
disable_pwm4_interrupt (***interrupt***)
disable_pwm5_interrupt (***interrupt***)
disable_pwm6_interrupt (***interrupt***)

**Parameters:**
***interrupt*** - 8-bit constant or variable. Constants are defined in the device's header file as:

pwm_period_interrupt
pwm_duty_interrupt
pwm_phase_interrupt
pwm_offset_interrupt

**Returns:**
Undefined

**Function:**
Disables one of the above PWM interrupts, multiple interrupts can be disabled by or'ing multiple options together.

**Availability:**
Devices with a 16-bit PWM module

**Requires:**
---

**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);
```

**See Also:**
setup_pwm(), set_pwm_duty(), set_pwm_phase(), set_pwm_period(), set_pwm_offset(), enable_pwm_interrupt(), clear_pwm_interrupt(), pwm_interrupt_active()

## div( ) ldiv( )

**Syntax:**
idiv=div(***num***, ***denom***)
ldiv =ldiv(***lnum***, ***ldenom***)

**Parameters:**
***num*** and ***denom*** are signed integers.
***num*** is the numerator and ***denom*** is the denominator
***lnum*** and ***ldenom*** are signed longs
*[PCD]* l***num*** and ***ldenom*** are signed int32, int48 or int64
***lnum*** is the numerator and ***ldenom*** is the denominator

**Returns:**
idiv is a structure of type **div_t** and **lidiv** is a structure of type **ldiv_t**. The **div** function returns a structure of type **div_t**, comprising of both the quotient and the remainder. The **ldiv** function returns a structure of type **ldiv_t**, comprising of both the quotient and the remainder.

**Function:**
**idiv** is a structure of type **div_t** and **lidiv** is a structure of type **ldiv_t**. The **div** function returns a structure of type **div_t**, comprising of both the quotient and the remainder. The **ldiv** function returns a structure of type **ldiv_t**, comprising of both the quotient and the remainder.

**Availability:**
All Devices

**Requires:**
#INCLUDE <STDLIB.H>

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

lidiv=ldiv(300,250);         //lidiv will contain lidiv.quot=1
and lidiv.rem=5
```

## dma_start( )

**Syntax:**
dma_start(***channel, mode, destAddr, sourceAddr, destCount, sourceCount***);
*[PCD]* dma_start(***channel***, ***mode***, ***addressA, addressB, count***);
*[PCD]* dma_start(***channel***, ***mode***, **destAddr, sourceAddr*, count***);

**Parameters:**
***Channel*** - The DMA channel to use.

***mode*** - The mode to use for the DMA transfers. Constants for setting the mode are defined in the device's header file. See the header file for all possible options.

***destAddr*** - The start RAM address of the destination address to use, can be anywhere in the GPR or SFR memory areas.

***sourceAddr*** - The start address of the source address to use, can be anywhere in RAM, EEPROM or Flash program memory areas. The memory area of the address reads from is determined by one of the settings that can be made with the mode parameter.

***destCount*** - The number of bytes to transfer to the destination address.

***sourceCount*** - The number of bytes to transfer from the source address for each DMA trigger.

*[PCD]* ***addressA*** - The start RAM address of the buffer to use located within the DMA RAM bank.

*[PCD]* ***addressB*** - If using DMA_PING_PONG mode the start RAM address of the second buffer to use located within the DMA RAM bank.

*[PCD]* ***destAddr*** - The start RAM address of the destination address to use, located within the DMA RAM bank. Address data is moved from.

*[PCD]* ***sourceAddr*** - The start RAM address of the source address to use, located within the DMA RAM bank. Address data is moved to.

*[PCD]* ***count -*** The number of DMA transfers to do. For devices with Type 1 DMA peripheral, this should be one less the actual number of transfers to do. For devices with Type 2 DMA peripheral, this should be equal to the actual number of transfers to do.

**Returns:**
Void

**Function:**
Starts the DMA transfer for the specified channel in the specified mode of operation and assigns the RAM addresses to use the DMA transfer.

**Availability:**
Devices that have the DMA peripheral. The version of the function used depends on the type of DMA peripheral it has. Use ***getenv("DMA")*** to determine the type the device has. *[PCD]* It will return 0 for no DMA peripheral, 1 for Type 1 and 2 for Type 2. For devices with Type 1 uses first version of the function and for devices with Type 2 uses second version of the function.

**Requires:**
---

**Examples:**
```
dma_start(1, DMA_SOURCE_ADDR_IS_SFR_GPR |
DMA_SOURCE_ADDR_UNCHANGED |
DMA_INC_DEST_ADDR | DMA_HW_TRIGGER_STARTS_XFER | DMA_CONTINUOUS,
RxBuffer, getenv("SFR:U1RXB"), DMA_BUFFER_SIZE, 1);

[PCD]
dma_start(0,DMA_PING_PONG|DMA_CONTINUOUS, RxBuffer[0],
RxBuffer[1],(DMA_BUFFER_SIZE-1));                        // Type
```

```
1

dma_start(0,DMA_SOURCE_ADDR_UNCHANGED|DMA_INC_DEST_ADDR|
DMA_REPEATED|DMA_ONE_SHOT,RxBuffer,getenv("SFR:U1RXREG"),
DMA_BUFFER_SIZE);                                          // Type
2
```

**Example Files:**
ex_dma_uart_rx.c

**See Also:**
setup_dma(), dma_status()

## dma_status( )

**Syntax:**
Value = dma_status(***channel***);

**Parameters:**
***Channel –*** The channel in which the status is to be queried.

**Returns:**
The DMA channel's status. See the device header file for mask values that can be AND'ed with return value.

**Function:**
This function will return the status of the specified channel in the DMA module.

**Availability:**
Devices that have the DMA module

**Requires:**
---

**Examples:**

```
Int8 value;
value = dma_status(3);  // This will return the status of
channel 1 of the DMA module
```

**See Also:**
setup_dma(), dma_start()

## dmt_status( )

**Syntax:**
Status = dmt_status();

**Parameters:**
---

**Returns:**
An int8 value indicating the status of the DT peripheral. See the device's header file for constants that can be and'ed with the return value to determine the state of the individual status bits.

**Function:**
Used to determine the status of the Deadman Timer (DMT) peripheral.

**Availability:**
Only on devices that have the DMT peripheral.

**Requires:**
---

**Examples:**
```
  if((dmt_status() & DMT_CLEAR_WINDOW_OPEN) == DMT_CLEAR_WINDOW_OPEN)
       clear_dmt();
```

**See Also:**
clear_dmt(), read_dmt(), disable_dmt(), enable_dmt(), setup_dmt()

## enable_dmt( )

**Syntax:**
enable_dmt();

**Parameters:**
---

**Returns:**
---

**Function:**
Enable the Deadman Timer (DMT). This function only works if the DMT_SW configuration fuses has been set. If the DMT configuration fuse is set, then the DMT is always enabled.

**Availability:**
Only on devices that have the DMT peripheral.

**Requires:**
---

**Examples:**
```
enabled_dmt();
```

**See Also:**
clear_dmt(), read_dmt(), disable_dmt(), dmt_status(), setup_dmt()

## enable_interrupts( )

**Syntax:**
enable_interrupts (***level***)
[PCD] enable_interrupts (***name***)
[PCD] enable_interrupts (***INTR_XX***)

**Parameters:**
***level* -** is a constant defined in the devices *.h file

[PCD] **name**- a constant defined in the devices .h file

[PCD] **INTR_XX** – Allows user selectable interrupt options like **intr_normal, intr_alternate, intr_level**

**Returns:**
Undefined

**Function:**
This 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 specified interrupts previously enabled to become active. Some chips that have an interrupt on change for individual pins all the pin to be specified, such as INT_RA1. For interrupts that use edge detection to trigger, it can be setup in the enable_interrupts( ) function without making a separate call to the set_int_edge( ) function.

Enabling interrupts does not clear the interrupt flag if there was a pending interrupt prior to the call. Use the clear_interrupt( ) function to clear pending interrupts before the call to enable_interrupts( ) to discard the prior interrupts.

*[PCD]*
**name** -Enables the interrupt for the given name. Valid specific names are the same as are used in #INT_xxx and are listed in the devices .h file.

**intr_global** – Enables all interrupt levels (same as INTR_LEVEL0)

**intr_normal** – Use normal vectors for the ISR

**intr_alternate** – Use alternate vectors for the ISR

**intr_level0.... intr_level7** – Enables interrupts at this level and above, interrupts at lower levels are disabled

**intr_cn_pin | pin_xx** – Enables a CN pin interrupts

**Availability:**
Devices that have interrupts and all 24-bit devices.

**Requires:**
Should have a #INT_XXXX to define the ISR, and constants are defined in the devices **.h* file.

**Examples:**
```
enable_interrupts(GLOBAL);
enable_interrupts(INT_TIMER0);
enable_interrupts( INT_EXT_H2L )

[PCD]
enable_interrupts(INT_TIMER0);
enable_interrupts(INT_TIMER1);
enable_interrupts(INTR_CN_PIN|Pin_B0);
```

**Example Files:**
ex_sisr.c, ex_stwt.c

**See Also:**
disable interrupts(), clear_interrupt (), ext_int_edge( ), #INT_xxxx, Interrupts Overview, interrupt_active()

## erase_program_memory( )

**Syntax:**
erase_program_memory (**address**);

**Parameters:**
**address** - is 32 bits. The least significant bits may be ignored.

**Returns:**
Undefined

**Function:**
Erases FLASH_ERASE_SIZE bytes to 0xFFFF in program memory.
FLASH_ERASE_SIZE varies depending on the part.

| Family | FLASH_ERASE_SIZE |
|---|---|
| dsPIC30F | 32 instructions (96 bytes) |
| dsPIC33FJ | 512 instructions (1536 bytes) |
| PIC24FJ | 512 instructions (1536 bytes) |
| PIC24HJ | 512 instructions (1536 bytes) |

NOTE: Each instruction on the PCD is 24 bits wide (3 bytes)
See write_program_memory() for more information on program memory access.

**Availability:**
All Devices

**Requires:**
---

**Examples:**

```
Int32 address = 0x2000;

erase_program_memory(address);          // erase block of memory
from 0x2000
                                        // to 0x2400 for a
PIC24HJ/FJ /33FJ
                                       //device, or erase 0x2000
to 0x2040
                                       //for a dsPIC30F chip
```

**See Also:**
write program eeprom() , write program memory(), Program Eeprom Overview

## enable_pwm1_interrupt( ) enable_pwm2_interrupt( ) enable_pwm3_interrupt( ) enable_pwm4_interrupt( ) enable_pwm5_interrupt( ) enable_pwm6_interrupt( )

**Syntax:**
enable_pwm1_interrupt (***interrupt***)
enable_pwm2_interrupt (***interrupt***)
enable_pwm3_interrupt (***interrupt***)
enable_pwm4_interrupt (***interrupt***)
enable_pwm5_interrupt (***interrupt***)
enable_pwm6_interrupt (***interrupt***)

**Parameters:**
***interrupt*** - 8-bit constant or variable. Constants are defined in the device's header file as:

PWM_PERIOD_INTERRUPT
PWM_DUTY_INTERRUPT
PWM_PHASE_INTERRUPT
PWM_OFFSET_INTERRUPT

**Returns:**
---

**Function:**
Enables one of the above PWM interrupts, multiple interrupts can be enabled by or'ing multiple options together. For the interrupt to occur, the overall PWMx interrupt still needs to be enabled and an interrupt service routine still needs to be created.

**Availability:**
Devices with a 16-bit PWM module.

**Requires:**
---

**Examples:**
```
enable_pwm1_interrupt(PWM_PERIOD_INTERRUPT);
enable_pwm1_interrupt(PWM_PERIOD_INTERRUPT |
PWM_DUTY_INTERRUPT);
```

**See Also:**
setup_pwm(), set_pwm_duty(), set_pwm_phase(), set_pwm_period(), set_pwm_offset(), disable_pwm_interrupt(), clear_pwm_interrupt(), pwm_interrupt_active()

## erase_eeprom( )

**Syntax:**
erase_eeprom (address);

**Parameters:**
**address** is 8 bits on PCB parts

**Returns:**
Undefined

**Function:**
This will erase a row of the EEPROM or Flash Data Memory.

**Availability:**
PCB devices with EEPROM like the 12F519

**Requires:**
---

**Examples:**
```
  erase_eeprom(0);        // erase the first row of the EEPROM (8 bytes)
```

**See Also:**
write_eeprom(), read_eeprom(), Data EEPROM Overview

## erase_program_memory( )

**Syntax:**
erase_program_eeprom (***address***);

**Parameters:**
***address -*** is 16 on PCM parts and 32 bits on PCH parts. The least significant bits may be ignored.
*[PCD]* ***address -*** is 32 bits. The least significant bits may be ignored.

**Returns:**
---

**Function:**
Erases FLASH_ERASE_SIZE bytes to 0xFFFF in program memory.
FLASH_ERASE_SIZE varies depending on the part. For example, if it is 64 bytes then the least significant 6 bits of address is ignored.

[PCD] For example, if it is 128 bytes then the least significant 7 bits of address is ignored.

See write_program_memory() [PCD] EEPROM Overview for more information on program memory access.

**Availability:**
Only devices that allow writes to program memory.

**Requires:**
---

**Examples:**
```
for(i=0x1000;i<=0x1fff;i+=getenv("FLASH_ERASE_SIZE"))
erase_program_memory(i);
```

**See Also:**
write program eeprom(), write program memory(), Program Eeprom Overview

## exp( )

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

**Parameters:**
***value* is** a float
[PCD] ***value*** is any float type

**Returns:**
A float
[PCD] ***A*** float with a precision equal to ***value***

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

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

Range error occur in the following case: **exp** when the argument is too large

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

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

**See Also:**
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.
*[PCD]* **source** is a constant from 0 to 4.
Source is optional and defaults to 0.

***edge*** is a constant H_TO_L or L_TO_H representing "high to low" and "low to high"

**Returns:**
Undefined

**Function:**
Determines when the external interrupt is acted upon. The edge may be L_TO_H or H_TO_L to specify the rising or falling edge.

**Availability:**
Only devices with interrupts

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

**Examples:**
```
  ext_int_edge( 2, L_TO_H);     // Set up PIC18 EXT2
  ext_int_edge( 2, L_TO_H);     // Set up external interrupt 2 to
  interrupt
                                // on rising edge
  ext_int_edge( H_TO_L );       // Sets up EXT
```

```
ext_int_edge( H_TO_L );       // Sets up external interrupt 0 to
interrupt
                              // on falling edge
```

**Example Files:**
ex_wakup.c

**See Also:**
#INT_EXT , enable_interrupts() , disable_interrupts() , #INT_EXT , enable_interrupts() , disable_interrupts , Interrupts Overview

## fabs( )

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

**Parameters:**
***value*** is a float
*[PCD]* ***value*** is any float type

**Returns:**
result is a float
*[PCD]* ***r***esult is a float with precision to ***value***

**Function:**
The **fabs()** function computes the absolute value of a float

**Availability:**
All Devices

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

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

**See Also:**
abs(), labs()

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

**Syntax:**
value = getc()

value = fgetc(***stream***)
value=getch()
value=getchar()

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

**Returns:**
An 8-bit character

**Function:**
This function waits for a character to come in over the RS232 RCV pin and returns the character.
In order to not 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 device.

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

**Availability:**
All Devices

**Requires:**
#USE RS232

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

#use rs232(baud=9600,xmit=pin_c6,
           rcv=pin_c7,stream=HOSTPC)
#use rs232(baud=1200,xmit=pin_b1,
            rcv=pin_b0,stream=GPS)
#use rs232(baud=9600,xmit=pin_b3,
            stream=DEBUG)
...
while(TRUE) {
   c=fgetc(GPS);
   fputc(c,HOSTPC);
   if(c==13)
```

```
      fprintf(DEBUG,"Got a CR\r\n");
}
```

**Example Files:**
ex_stwt.c

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

## gets( ) fgets( )

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

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

**Returns:**
Undefined

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

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

**Availability:**
All Devices

**Requires:**
#USE RS232

**Examples:**

```
char string[30];

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

**See Also:**
getc(), get_string in input.c

## floor( )

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

**Parameters:**
***value is*** a float
*[PCD]* ***value*** is any float type

**Returns:**
Result is a float
*[PCD]* ***R***esult is a float with precision equal to ***value***

**Function:**
Computes the greatest integer value not greater than the argument. Floor (12.67) is 12.00

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

**Examples:**

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

**See Also:**
ceil()

## fmod( )

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

**Parameters:**
***val1*** is a float
*[PCD]* ***val1*** is any float type
***val2*** is a float
*[PCD]* ***val2*** is any float type

**Returns:**
Result is a float
*[PCD]* ***R***esult is a float with precision equal to input parameters ***val1*** and ***val2***

**Function:**
Returns the floating point remainder of val1/val2. Returns the value val1 - i*val2 for some integer "i" such that, if val2 is nonzero, the result has the same sign as val1 and magnitude less than the magnitude of val2.

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

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

## printf( ) fprintf( )

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

**Parameters:**
***String*** is a constant string or an array of characters null terminated.

***C String*** is a constant string. Note that format specifiers cannot be used in RAM strings.

***Values*** is a list of variables separated by commas, fname is a function name to be used for outputting (default is **putc** is none is specified.

***Stream*** is a stream identifier (a constant byte)

**Returns:**
Undefined

**Function:**
Outputs a string of characters to either the standard RS-232 pins (first two forms) or to a specified function. Formatting is in accordance with the string argument. When variables are used this string must be a constant. The % character is used within the string to indicate a variable value is to be formatted and output. Longs in the printf may be 16 or 32 bit. A %% will output a single %. Formatting rules for the % follows.

See the Expressions > Constants and Trigraph sections of this manual for other escape character that may be part of the string.

If **fprintf()** is used then the specified stream is used where **printf()** defaults to STDOUT (the last USE RS232).

Format:
The format takes the generic form %nt. n is optional and may be 1-9 to specify how many characters are to be outputted, or 01-09 to indicate leading zeros, or 1.1 to 9.9 for floating point and %w output. t is the type and may be one of the following:

**c** -- string or character
**u** -- unsigned
**d** -- signed int
**Lu --** long unsigned int
**Ld --** long signed int
**x --** hex int (lower case)
**X --** hex int (upper case
**Lx** -- hex long int (lower case)
**LX** -- hex long int (upper case)
**f --** float with truncated decimal
**g --** float with rounded decimal
**e --** float in exponential format
**w --** unsigned int with decimal place inserted. Specify two numbers for ***n***. The first is a total field width. The second is the desired number of decimal places.

Example Formats:

| **Specifier** | **Value=0x12** | **Value=0xfe** |
|---|---|---|
| %03u | 018 | 254 |
| %u | 18 | 254 |
| %2u | 18 | * |
| %5 | 18 | 254 |
| %d | 18 | -2 |
| %x | 12 | fe |
| %X | 12 | FE |

| | | |
|---|---|---|
| %4X | 0012 | 00FE |
| %3.1w | 1.8 | 25.4 |

* Result is undefined - Assume garbage.

**Availability:**
All Devices

**Requires:**
#USE RS232 (unless fname is used)

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

**Example Files:**
ex_admm.c, ex_lcdkb.c

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

## putc( ) putchar( ) fputc( )

**Syntax:**
putc (***cdata***)
putchar (***cdata***)
fputc(***cdata***, ***stream***)

**Parameters:**
***cdata*** is a 8 bit character.
***Stream*** is a stream identifier (a constant byte)

**Returns:**
Undefined

**Function:**
This function sends a character over the RS232 XMIT pin. A #USE RS232 must appear before this call to determine the baud rate and pin used. The #USE RS232 remains in effect until another is encountered in the file.

If This function sends a character over the RS232 XMIT pin. A #USE RS232 must appear before this call to determine the baud rate and pin used. The #USE RS232 remains in effect until another is encountered in the file.

If **fputc()** is used then the specified stream is used where **putc()** defaults to STDOUT (the last USE RS232). is used then the specified stream is used where **putc()** defaults to STDOUT (the last USE RS232).

**Availability:**
All Devices

**Requires:**
#USE RS232

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

**Example Files:**
ex_tgetc.c

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

## puts( ) fputs( )

**Syntax:**
puts (***string***).
fputs (***string***, ***stream***)

**Parameters:**
***string*** is a constant string or a character array (null-terminated).
***stream*** is a stream identifier (a constant byte)

**Returns:**
Undefined

**Function:**
Sends each character in the string out the RS232 pin using **putc()**. After the string is sent a CARRIAGE-RETURN (13) and LINE-FEED (10) are sent. In general **printf()** is more useful than **puts()**.

If **fputs()** is used then the specified stream is used where **puts()** defaults to STDOUT (the last USE RS232)

**Availability:**
All Devices

**Requires:**
#USE RS232

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

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

## free( )

**Syntax:**
free(***ptr***)

**Parameters:**
***ptr is a pointer earlier returned by the calloc, malloc or realloc***

**Returns:**
No Value

**Function:**
The free function causes the space pointed to by the ***ptr*** to be deallocated, that is made available for further allocation. If ***ptr*** is a null pointer, no action occurs. If the ***ptr*** does not match a pointer earlier returned by the **calloc**, **malloc** or **realloc**, or if the space has been deallocated by a call to free or **realloc** function, the behavior is undefined.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlibm.h>

**Examples:**
```
int * iptr;
iptr=malloc(10);
```

```
free(iptr)                // iptr will be deallocated
```

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

## frexp( )

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

**Parameters:**
***value*** is a float
*[PCD]* ***value*** is any float type

***exp*** is a signed int

**Returns:**
result is a float
*[PCD]* result is a float with precision equal to ***value***

**Function:**
The **frexp** function breaks a floating point number into a normalized fraction and an integral power of 2. It stores the integer in the signed **int** object **exp**. The result is in the interval [1/2 to1) or zero, such that value is result times 2 raised to power **exp**. If value is zero then both parts are zero.

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

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

**See Also:**
ldexp(), exp(), log(), log10(), modf()

## scanf( ) fscanf( )

**Syntax:**
scanf(**cstring**);
scanf(**cstring, values...**)
fscanf(**stream, cstring, values...**

**Parameters:**
***cstring*** is a constant string.

**values** is a list of variables separated by commas.

**stream** is a stream identifier

**Returns:**
0 if a failure occurred, otherwise it returns the number of conversion specifiers that were read in, plus the number of constant strings read in.

**Function:**
Reads in a string of characters from the standard RS-232 pins and formats the string according to the format specifiers. The format specifier character (%) used within the string indicates that a conversion specification is to be done and the value is to be saved into the corresponding argument variable. A %% will input a single %. Formatting rules for the format specifier as follows:

If fscanf() is used, then the specified stream is used, where scanf() defaults to STDIN (the last USE RS232).

Format:
The format takes the generic form %nt. **n** is an option and may be 1-99 specifying the field width, the number of characters to be inputted. **t** is the type and maybe one of the following:

**c** Matches a sequence of characters of the number specified by the field width (1 if no field width is specified). The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence.

**s** Matches a sequence of non-white space characters. The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence and a terminating null character, which will be added automatically.

**u** Matches an unsigned decimal integer. The corresponding argument shall be a pointer to an unsigned integer.

**Lu** Matches a long unsigned decimal integer. The corresponding argument shall be a pointer to a long unsigned integer.

**d** Matches a signed decimal integer. The corresponding argument shall be a pointer to a signed integer.

**Ld** Matches a long signed decimal integer. The corresponding argument shall be a pointer to a long signed integer.

**o** Matches a signed or unsigned octal integer. The corresponding argument shall be a pointer to a signed or unsigned integer.

**Lo** Matches a long signed or unsigned octal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.

**x or X** Matches a hexadecimal integer. The corresponding argument shall be a pointer to a signed or unsigned integer.

**Lx or LX** Matches a long hexadecimal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.

**i** Matches a signed or unsigned integer. The corresponding argument shall be a pointer to a signed or unsigned integer.

**Li** Matches a long signed or unsigned integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.

**f,g or e** Matches a floating point number in decimal or exponential format. The corresponding argument shall be a pointer to a float.

**[** Matches a non-empty sequence of characters from a set of expected characters. The sequence of characters included in the set are made up of all character following the left bracket ([) up to the matching right bracket (]). Unless the first character after the left bracket is a **^**, in which case the set of characters contain all characters that do not appear between the brackets. If a **-** character is in the set and is not the first or second, where the first is a **^**, nor the last character, then the set includes all characters from the character before the **-** to the character after the **-**.

For example, %[a-z] would include all characters from **a** to **z** in the set and %[^a-z] would exclude all characters from **a** to **z** from the set. The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence and a terminating null character, which will be added automatically.

**n** Assigns the number of characters read thus far by the call to scanf() to the corresponding argument. The corresponding argument shall be a pointer to an unsigned integer.

An optional assignment-suppressing character (*****) can be used after the format specifier to indicate that the conversion specification is to be done, but not saved into a corresponding variable. In this case, no corresponding argument variable should be passed to the scanf() function.

A string composed of ordinary non-white space characters is executed by reading the next character of the string. If one of the inputted characters differs from the string, the function fails and exits. If a white-space character precedes the ordinary non-white space characters, then white-space characters are first read in until a non-white space character is read.

White-space characters are skipped, except for the conversion specifiers **[**, **c** or **n**, unless a white-space character precedes the **[** or **c** specifiers.

**Availability:**
All Devices

**Requires:**
#USE RS232

**Examples:**
```
char name[2-];
unsigned int8 number;
signed int32 time;

if(scanf("%u%s%ld",&number,name,&time))
   printf"\r\nName: %s, Number: %u, Time: %ld",name,number,time
```

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

## get_adc_ports( )

**Syntax:**
value = get_adc_ports();

**Parameters:**
---

**Returns:**
A 32-bit int

**Function:**
Returns a value that can be passed to setup_adc_ports() to setup the analog pins.

**Availability:**
Devices with an Analog-to-Digital (ADC) module.

**Requires:**
---

**Examples:**

```
adc_pins = get_adc_ports();
```

**See Also:**
read_adc( ), setup_adc(mode), setup_adc_ports( ), set_adc_channel( ), ADC

## get_capture( )

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

**Parameters:**
***x*** defines which ccp module to read from

**Returns:**
A 16-bit timer value

**Function:**
This function obtains the last capture time from the indicated CCP module.

**Availability:**
Only available on devices with Input Capture modules

**Requires:**
---

**Example Files:**
ex_ccpmp.c

**See Also:**
setup_ccpx( )

## [PCD] get_capture( )

**Syntax:**
value = get_capture(***x***, ***wait***)

**Parameters:**
***x*** defines which input capture result buffer module to read from

***wait*** signifies if the compiler should read the oldest result in the buffer or the next result to enter the buffer

**Returns:**
A 16-bit timer value

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

**Availability:**
Only available on devices with Input Capture modules

**Requires:**
---

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

**See Also:**
setup_capture( ), setup_compare( ), Input Capture Overview

## get_capture32_ccp1( ) get_capture_ccp1( ) get_capture_ccp2() get_capture_ccp3( ) get_capture_ccp4( ) get_capture_ccp5( )

**Syntax:**
**value**=get_capture_ccpx(**wait**);

**Parameters:**
**wait** -signifies if the compiler should read the oldest result in the buffer or the next result in the buffer or the next result to enter the buffer

**Returns:**
**value16** -a 16-bit timer value

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

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

**Requires:**
---

**Examples:**
```
unsigned int16 value;

setup_ccp1(CCP_CAPTURE_FE);

while(TRUE) {
   value=get_capture_ccp1(TRUE);
   printf("Capture occurred at: %LU", value);
}
```

**See Also:**
set_pwmX_duty(), setup_ccpX(), set_ccpX_compare_time(), set_timer_ccpX(), set_timer_period_ccpX(), get_timer_ccpx(), get_capture32_ccpX()

## [PCD] get_capture32_ccp1( ) get_capture32_ccp2( ) get_capture32_ccp3( ) get_capture32_ccp4( ) get_capture32_ccp5( )

**Syntax:**
**value**=get_capture32_ccpx(**wait**);

**Parameters:**
**wait** -signifies if the compiler should read the oldest result in the buffer or the next result in the buffer or the next result to enter the buffer

**Returns:**
**value32** -a 32-bit timer value

**Function:**
If **wait** is true, the current capture values in the result buffer are cleared, and the next result to be sent, the buffer is returned. If **wait** is false, the default setting, the first value currently in the buffer is return. However, the buffer will only hold two results while waiting for them to be read. If read is not being called for every capture event, when **wait** is false, the buffer will fill with old capture values and any new result will be lost.

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

**Requires:**
---

**Examples:**
```
unsigned int32 value;

setup_ccp1(CCP_CAPTURE_FE|CCP_TIMER_32_BIT);

while(TRUE) {
   value=get_capture_ccp1(TRUE);
   printf("Capture occurred at: %LU", value);
}
```

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

## get_capture_event( )

**Syntax:**
result = get_capture_event([**stream**]);

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

**Returns:**
TRUE if a capture event occurred, FALSE otherwise

**Function:**
To determine if a capture event occurred.

**Availability:**
All Devices

**Requires:**
#USE CAPTURE

**Examples:**

```
#USE CAPTURE(INPUT=PIN_C2,CAPTURE_RISING,TIMER=1,FASTEST)
if(get_capture_event())
   result = get_capture_time()
```

**See Also:**
#use_capture, get_capture_time()

## get_capture_time( )

**Syntax:**
result = get_capture_time([stream])

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

**Returns:**
An int16 value representing the last capture time

**Function:**
To get the last capture time.

**Availability:**
All Devices

**Requires:**
#USE CAPTURE

**Examples:**
```
#USE CAPTURE(INPUT=PIN_C2,CAPTURE_RISING,TIMER=1,FASTEST)
result = get_capture_time();
```

**See Also:**
#use_capture, get_capture_event()

## [PCD] get_capture32( )

**Syntax:**
result = get_capture32(**x,[wait]**)

**Parameters:**
**x** is 1-16 and defines which input capture result buffer modules to read from.

**wait** is an optional parameter specifying if the compiler should read the oldest result in the bugger or the next result to enter the buffer

**Returns:**
A 32-bit timer value

**Function:**
If **wait** is true, the current capture values in the result buffer are cleared, and the next result to be sent to the buffer is returned. If **wait** is false, the default setting, the first value currently in the buffer is returned. However, the buffer will only hold four results while waiting for them to be read, so if get_capture32 is not being called for every capture event. When **wait** is false, the buffer will fill with old capture values and any new results will be lost.

**Availability:**
Only devices with a 32-bit Input Capture module

**Requires:**
---

**Examples:**
```
setup_timer2(TMR_INTERNAL | TMR_DIV_BY_1 | TMR_32_BIT);
setup_capture(1,CAPTURE_FE | CAPTURE_TIMER2 | CAPTURE_32_BIT);
while(TRUE) {
```

```
    timerValue=get_capture32(1,TRUE);
    printf("Capture 1 occurred at: %LU", timerValue);
}
```

**See Also:**
setup_capture(), setup_compare(), get_capture(), Input Capture Overview

## get_hspwm_capture( )

**Syntax:**
result=get_hspwm_capture(**unit**);

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

**Returns:**
Unsigned in16 value representing the capture PWM time base value.

**Function:**
Gets the captured PWM time base value from the leading edge detection on the current-limit input.

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

**Requires:**
---

**Examples:**
```
result=get_hspwm_capture(1);
```

**See Also:**
setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_blanking(), setup_hspwm_trigger(), set_hspwm_override(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock()
setup_hspwm(), setup_hspwm_secondary()

## get_hspwm_feedback( )

**Syntax:**
result = get_hspwm_feedback();

**Parameters:**
---

**Returns:**
A 15-bit pseudorandom value.

**Function:**
To read the Linear Feedback Shirt register from the High-Speed PWM (HSPWM) peripheral.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
unsigned int16 value;
value = get_hspwm_feedback();
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## get_hspwm_status( )

**Syntax:**
result = get_hspwm_status(**unit**);

**Parameters:**
***unit*** - The PWM unit to get the status for.

**Returns:**
A 16-bit value indicating the status of the specified PWM unit. See the device's header file for the defines that can be and'ed with return value to determine if the corresponding status bits are set.

**Function:**
To read the Linear Feedback Shirt register from the High-Speed PWM (HSPWM) peripheral.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
unsigned int16 status;
status = get_hspwm_status(1);
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## get_motor_pwm_count( )

**Syntax:**
Data16 = get_motor_pwm_count(pwm);

**Parameters:**
***pwm*- Defines the pwm module used**

**Returns:**
16 bits of data

**Function:**
Returns the PWM count of the motor control unit

**Availability:**
Devices that have the motor control PWM unit

**Requires:**
---

**Examples:**
```
  Data16 = get_motor_pmw_count(1)
```

**See Also:**
setup_motor_pwm(), set_motor_unit(), set_motor_pwm_event(), set_motor_pwm_duty()
.

## get_nco_accumulator( )

**Syntax:**
value =get_nco_accumulator( );

**Parameters:**
---

**Returns:**
Current value of accumulator

**Function:**
Returns the PWM count of  the motor control unit

**Availability:**
Devices that have a NCO module

**Requires:**
---

**Examples:**
```
  value=get_nco_accumulator();
```

**See Also:**
setup_nco( ), set_nco_inc_value( ), get_nco_inc_value( )
.

## get_nco_inc_value( )

**Syntax:**
value =get_nco_inc_value( );

**Parameters:**
---

**Returns:**
Current value set in increment registers

**Function:**
Returns the PWM count of the motor control unit

**Availability:**
Devices that have the motor control PWM unit

**Requires:**
---

**Examples:**
```
Data16 = get_motor_pmw_count(1)
```

**See Also:**
setup_nco( ), set_nco_inc_value( ), get_nco_accumulator( )

## get_ticks( )

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

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

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

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

**Availability:**
All Devices

**Requires:**
#USE TIMER(options)

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

void main(void) {
   unsigned int16 current_tick;

   current_tick = get_ticks();
}
```

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

## get_timerA( )

**Syntax:**
value=get_timerA();

**Parameters:**
---

**Returns:**
The current value of the timer as an int8

**Function:**
Returns the current value of the timer. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2, …).

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

**Requires:**
---

**Examples:**
```
set_timerA(0);
while(timerA < 200);
```

**See Also:**
set_timerA( ), setup_timer_A( ), TimerA Overview

## get_timerB( )

**Syntax:**
value=get_timerB();

**Parameters:**
---

**Returns:**
The current value of the timer as an int8

**Function:**
Returns the current value of the timer. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2, …).

**Availability:**
This function is only available on devices with Timer B hardware

**Requires:**
---

**Examples:**
```
set_timerB(0);
while(timerB < 200);
```

**See Also:**
set_timerB( ), setup_timer_B( ), TimerB Overview

## get_timerx( )

**Syntax:**
value=get_timer0() Same as: value=get_rtcc()
value=get_timer1()
value=get_timer2()
value=get_timer3()
value=get_timer4()
value=get_timer5()
value=get_timer6()
value=get_timer7()
value=get_timer8()
value=get_timer10()
value=get_timer12()
*[PCD]* value=get_timer1( )

[PCD] value=get_timer2( )
[PCD] value=get_timer3( )
[PCD] value=get_timer4( )
[PCD] value=get_timer5( )
[PCD] value=get_timer6( )
[PCD] value=get_timer7( )
[PCD] value=get_timer8( )
[PCD] value=get_timer9( )

**Parameters:**
---

**Returns:**
Timers 1, 3, 5 and 7 return a 16 bit int.
Timers 2 ,4, 6, 8, 10 and 12 return an 8 bit int.
Timer 0 (AKA RTCC) returns a 8 bit int except on the PIC18XXX where it returns a 16 bit int.
[PCD] The current value of the timer as an int16

**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...)
[PCD] Retrieves the value of the timer, specified by X (which may be 1-9)

**Availability:**
Timer 0 - All devices
Timers 1 & 2 - Most but not all PCM devices
Timer 3, 5 and 7 - Some PIC18 and Enhanced PIC16 devices
Timer 4,6,8,10 and 12- Some PIC18 and Enhanced PIC16 devices
[PCD] This function is available on all devices that have a valid timerX

**Requires:**
---

**Examples:**
```
set_timer0(0);
while ( get_timer0() < 200 ) ;
if(get_timer2() % 0xA0 == HALF_WAVE_PERIOD)
output_toggle(PIN_B0);
```

**Example Files:**
ex_stwt.c

**See Also:**
set_timerx(), Timer0 Overview , Timer1 Overview , Timer2 Overview , Timer5 Overview
*[PCD]* Timer Overview , setup_timerX(), get_timerXY(), set_timerX(), set_timerXY()

## get_timerxy( )

**Syntax:**
value=get_timer23( )
value=get_timer45( )
value=get_timer67( )
value=get_timer89( )

**Parameters:**
Void

**Returns:**
The current value of the 32 bit timer as an int32

**Function:**
Retrieves the 32 bit value of the timers X and Y, specified by XY (which may be 23, 45, 67 and 89)

**Availability:**
This function is available on all devices that have a valid 32 bit enabled timers. Timers 2 & 3, 4 & 5, 6 & 7 and 8 & 9 may be used. The target device must have one of these timer sets. The target timers must be enabled as 32 bit.

**Requires:**
---

**Examples:**
```
if(get_timer23() > TRIGGER_TIME)
   ExecuteEvent();
```

**Example Files:**
ex_stwt.c

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

## get_timer_ccp1( ) get_timer_ccp2( ) get_timer_ccp3( ) get_timer_ccp4( ) get_timer_ccp5( )

**Syntax:**
**value32**=get_timer_ccpx();
**value16**=get_timer_ccpx(**which**);

**Parameters:**
**which** - when in 16-bit mode determines which timer value to read. 0 reads the lower timer value (CCPxTMRL), and 1 reads the upper timer value (CCPxTMRH)

**Returns:**
**value32** - the 32-bit timer value.

**value16**- the 16-bit timer value

**Function:**
This function gets the timer values for the CCP module

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

**Requires:**
---

**Examples:**

```
unsigned int32 value32;
unsigned int32 value15;
value32=get_timer_ccpx();          //get the 32 bit timer value
value16=get_timer_ccpx(0);         //get the 16 bit timer value
from
                                   //lower timer
value16=get_timer_ccpx(1);         //get the 16 bit timer value
from
                                   //upper timer
```

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

## get_tris_x( )

**Syntax:**
value = get_tris_A();
value = get_tris_B();
value = get_tris_C();
value = get_tris_D();
value = get_tris_E();
value = get_tris_F();
value = get_tris_G();
value = get_tris_H();
value = get_tris_J();
value = get_tris_K();
value = get_tris_L();

**Parameters:**
---

**Returns:**
int16, the value of TRIS register

**Function:**
Returns the value of the TRIS register of port A, B, C, D, E, F, G, H, J, K or L.

**Availability:**
All Devices

**Requires:**
---

**Examples:**
```
tris_a = GET_TRIS_A()
```

**See Also:**
input(), output_low(), output_high()

## get_wdt( )

**Syntax:**
value = get_wdt();

**Parameters:**
---

**Returns:**
An 8-bit int

**Function:**
Returns the current watchdog timer value.

**Availability:**
Devices with a Windowed Watchdog Timer.

**Requires:**
---

**Examples:**

```
int8 count;
count = get_wdt();
```

**See Also:**
setup_wdt( ), restart_wdt( ), WDT or Watch Dog Timer

## getenv( )

**Syntax:**
value = getenv (***cstring****);*

**Parameters:**
**cstring -** is a constant string with a recognized keyword

**Returns:**
A constant number, a constant string or 0

**Function:**
This function obtains information about the execution environment. The following are recognized keywords. This function returns a constant 0 if the keyword is not understood.

| | |
|---|---|
| **FUSE_SET:fffff** | **Returns 1 if fuse fffff is enabled** |
| **FUSE_VALID:fffff** | Returns 1 if fuse fffff is valid |
| **INT:iiiii** | Returns 1 if the interrupt iiiii is valid |
| **ID** | Returns the device ID (set by #ID) |

| | |
|---|---|
| **DEVICE** | Returns the device name string (like "PIC16C74") |
| **CLOCK** | Returns the MPU FOSC |
| **VERSION** | Returns the compiler version as a float |
| **VERSION_STRING** | Returns the compiler version as a string |
| **PROGRAM_MEMORY** | Returns the size of memory for code (in words) |
| **STACK** | Returns the stack size |
| **SCRATCH** | Returns the start of the compiler scratch area |
| **DATA_EEPROM** | Returns the number of bytes of data EEPROM |
| **EEPROM_ADDRESS** | Returns the address of the start of EEPROM. 0 if not supported by the device. |
| **READ_PROGRAM** | Returns a 1 if the code memory can be read |
| **ADC_CHANNELS** | Returns the number of A/D channels |
| **ADC_RESOLUTION** | Returns the number of bits returned from READ_ADC() |
| **ICD** | Returns a 1 if this is being compiled for a ICD |
| **SPI** | Returns a 1 if the device has SPI |
| **USB** | Returns a 1 if the device has USB |
| **CAN** | Returns a 1 if the device has CAN |
| **I2C_SLAVE** | Returns a 1 if the device has I2C slave H/W |
| **I2C_MASTER** | Returns a 1 if the device has I2C master H/W |
| **PSP** | Returns a 1 if the device has PSP |
| **COMP** | Returns a 1 if the device has a comparator |

| | |
|---|---|
| **VREF** | Returns a 1 if the device has a voltage reference |
| **LCD** | Returns a 1 if the device has direct LCD H/W |
| **UART** | Returns the number of H/W UARTs |
| **AUART** | Returns 1 if the device has an ADV UART |
| **CCPx** | Returns a 1 if the device has CCP number x |
| **TIMERx** | Returns a 1 if the device has TIMER number x |
| **FLASH_WRITE_SIZE** | Smallest number of bytes that can be written to FLASH |
| **FLASH_ERASE_SIZE** | Smallest number of bytes that can be erased in FLASH |
| **BYTES_PER_ADDRESS** | Returns the number of bytes at an address location |
| **BITS_PER_INSTRUCTION** | Returns the size of an instruction in bits |
| **RAM** | Returns the number of RAM bytes available for your device. |
| **SFR:name** | Returns the address of the specified special file register. The output format can be used with the preprocessor command #bit. name must match SFR denomination of your target PIC (example: STATUS, INTCON, TXREG, RCREG, etc) |
| **BIT:name** | Returns the bit address of the specified special file register bit. The output format will be in “address:bit”, which can be used with the preprocessor command #byte. name must match SFR.bit denomination of your target PIC (example: C, Z, GIE, TMR0IF, etc) |
| **SFR_VALID:name** | Returns TRUE if the specified special file register name is valid and exists for your target PIC (example: getenv("SFR_VALID:INTCON")) |

| | |
|---|---|
| **BIT_VALID:name** | Returns TRUE if the specified special file register bit is valid and exists for your target PIC (example: getenv("BIT_VALID:TMR0IF")) |
| **PIN:PB** | Returns 1 if PB is a valid I/O PIN (like A2) |
| **UARTx_RX** | Returns UARTxPin (like PINxC7) |
| **UARTx_TX** | Returns UARTxPin (like PINxC6) |
| **SPIx_DI** | Returns SPIxDI Pin |
| **SPIxDO** | Returns SPIxDO Pin |
| **SPIxCLK** | Returns SPIxCLK Pin |
| **ETHERNET** | Returns 1 if device supports Ethernet |
| **QEI** | Returns 1 if device has QEI |
| **DAC** | Returns 1 if device has a D/A Converter |
| **DSP** | Returns 1 if device supports DSP instructions |
| **DCI** | Returns 1 if device has a DCI module |
| **DMA** | Returns 1 if device supports DMA |
| **CRC** | Returns 1 if device has a CRC module |
| **CWG** | Returns 1 if device has a CWG module |
| **NCO** | Returns 1 if device has a NCO module |
| **CLC** | Returns 1 if device has a CLC module |
| **DSM** | Returns 1 if device has a DSM module |
| **OPAMP** | Returns 1 if device has op amps |
| **RTC** | Returns 1 if device has a Real Time Clock |

| | |
|---|---|
| **CAP_SENSE** | Returns 1 if device has a CSM cap sense module and 2 if it has a CTMU module |
| **EXTERNAL_MEMORY** | Returns 1 if device supports external program memory |
| **INSTRUCTION_CLOCK** | Returns the MPU instruction clock |
| **ENH16** | Returns 1 for Enhanced 16 devices |
| *[PCD]* **ENH24** | Returns 2 for Enhanced 24 devices |
| *[PCD]* **IC** | Returns number of Input Capture units device has |
| *[PCD]* **ICx** | Returns TRUE if ICx is on this part |
| *[PCD]* **OC** | Returns number of Output Compare units device has |
| *[PCD]* **OCx** | Returns TRUE if OCx is on this part |
| *[PCD]* **RAM_START** | Returns the starting address of the first general purpose RAM location |
| *[PCD]* **PSV** | Returns TRUE if program space visibility (PSV) is enabled. If PSV is enabled, data in program memory ('const char *' or 'rom char *') can be assigned to a regular RAM pointer ('char *') and a regular RAM pointer can dereference data from program memory or RAM. |
| *[PCD]* **MIN_FLASH_WRITE** | The smallest number of bytes that can be written to FLASH using the write_program_memory() function. The write_program_memory() function can only write multiples of this size to the FLASH. Additionally, the start address passed to the write_program_memory() function must be multiples of this value divided by two. For example, if MIN_FLASH_WRITE is 4, then start address can be 0x0000, 0x0002, 0x004, etc. |

**Availability:**
All Devices

**Requires:**
---

**Examples:**
```
  #IF  getenv("VERSION")<3.050
     #ERROR  Compiler version too old
  #ENDIF

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

  #IF getenv("FUSE_VALID:BROWNOUT")
     #FUSE BROWNOUT
  #ENDIF

  #byte status_reg=GETENV("SFR:STATUS")
  #bit carry_flag=GETENV("BIT:C")
```

## goto_address( )

**Syntax:**
goto_address(***location***);

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

**Returns:**
---

**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:**
---

**Examples:**
```
  #define LOAD_REQUEST PIN_B1
  #define LOADER 0x1f00
```

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

**Example Files:**
setjmp.h

**See Also:**
label_address( )

## high_speed_adc_done( )

**Syntax:**
value = high_speed_adc_done([***pair***]);

**Parameters:**
**pair –** Optional parameter that determines which ADC pair's ready flag to check. If not used all ready flags are checked

**Returns:**
An int16. If pair is used 1 will be return if ADC is done with conversion, 0 will be return if still busy. If pair is not used, it will return a bit map of which conversion are ready to be read.
For example a return value of 0x0041 means that ADC pair 6, AN12 and AN13, and ADC pair 0, AN0 and AN1, are ready to be read.

**Function:**
Can be polled to determine if the ADC has valid data to be read.

**Availability:**
Only on dsPIC33FJxxGSxxx devices

**Requires:**
---

**Examples:**
```
int16 result[2]
setup_high_speed_adc_pair(1,  INDIVIDUAL_SOFTWARE_TRIGGER);
setup_high_speed_adc( ADC_CLOCK_DIV_4);

read_high_speed_adc(1, ADC_START_ONLY);
while(!high_speed_adc_done(1));
read_high_speed_adc(1, ADC_READ_ONLY, result);
```

```
printf("AN2 value = %LX, AN3 value =
%LX\n\r",result[0],result[1])
```

**See Also:**
setup_high_speed_adc(), setup_high_speed_adc_pair(), read_high_speed_adc()

## hspwm_do_capture( )

**Syntax:**
hspwm_do_capture(**unit**);

**Parameters:**
***unit*** - The PWM unit to capture.

**Returns:**
---

**Function:**
To initiate a software capture for the specified PWM unit of the High-Speed PWM (HSPWM) peripheral. Once a capture event has occurred, no further captures will occur until it is cleared. The capture event is cleared when read with the get_hspwm_capture() function.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
hspwm_do_capture(1);     //initiate a software capture for PWM unit 1
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_update()

## hspwm_stop_pwm( )

**Syntax:**
hspwm_stop_pwm(**unit**);

**Parameters:**
***unit*** - The PWM unit to stop.

**Returns:**
---

**Function:**
To stop a PWM cycle for the specified unit of the High-Speed PWM (HSPWM) peripheral.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
hspwm_stop_pwm(1);      // stop a PWM cycle for PWM unit 1
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_do_capture(), hspwm_update()

## hspwm_trigger_pwm( )

**Syntax:**
hspwm_trigger_pwm(**unit**);

**Parameters:**
***unit*** - The PWM unit to trigger.

**Returns:**
---

**Function:**
To trigger a PWM cycle for the specified unit of the High-Speed PWM (HSPWM) peripheral.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
  hspwm_trigger_pwm(1));     //trigger a software capture for PWM unit 1
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## hspwm_update( )

**Syntax:**
hspwm_update(**unit**);

**Parameters:**
***unit*** - The PWM unit to update.

**Returns:**
---

**Function:**
To request an update of the PWM data registers for the specified PWM unit of the High-Speed PWM (HSPWM) peripheral. When an update is pending, the UPDATE status bits will be set and the get_hspwm_status() function can be used to determine if an update is pending.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
hspwm_update(1);       // request an update for PWM unit 1
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(),

## i2c_init( )

**Syntax:**
i2c_init([stream],baud);

**Parameters:**
**stream** – optional parameter specifying the stream defined in #USE I2C.

**baud –** if baud is 0, I2C peripheral will be disable. If baud is 1, I2C peripheral is initialized and enabled with baud rate specified in #USE I2C directive. If baud is > 1 then I2C peripheral is initialized and enabled to specified baud rate

**Returns:**
---

**Function:**
To initialize I2C peripheral at run time to specified baud rate.

**Availability:**
All Devices

**Requires:**
#USE I2C

**Examples:**

```
#USE I2C(MASTER,I2C1, FAST,NOINIT)
i2c_init(TRUE);                          //initialize and enable
I2C peripheral
                                         //to baud rate specified
in //#USE I2C
i2c_init(500000);                        //initialize and enable
I2C peripheral
                                         //to a baud rate of 500
KBPS
```

**See Also:**
i2c_poll( ), i2c_speed( ), i2c_slaveaddr( ), i2c_isr_state( ) ,i2c_write( ), i2c_read( ), _use_i2c( ), i2c( )

## i2c_isr_state( )

**Syntax:**
state = i2c_isr_state();
state = i2c_isr_state(**stream**);

**Parameters:**
---

**Returns:**
**state -** is an 8 bit int

**0** - Address match received with R/W bit clear, perform **i2c_read( )** to read the I2C address.

**1-0x7F** - Master has written data; **i2c_read()** will immediately return the data

**0x80** - Address match received with R/W bit set; perform **i2c_read( )** to read the I2C address, and use **i2c_write( )** to pre-load the transmit buffer for the next transaction (next I2C read performed by master will read this byte).

**0x81-0xFF** - Transmission completed and acknowledged; respond with **i2c_write()** to pre-load the transmit buffer for the next transition (the next I2C read performed by master will read this byte).

**Function:**
Returns the state of I2C communications in I2C slave mode after an SSP interrupt. The return value increments with each byte received or sent.

If 0x00 or 0x80 is returned, an i2C_read( ) needs to be performed to read the I2C address that was sent (it will match the address configured by #USE I2C so this value can be ignored)

**Availability:**
Devices with built-in I2C

**Requires:**
#USE I2C

**Examples:**

```
#INT_SSP
   void i2c_isr() {
      state = i2c_isr_state();
      if(state== 0 ) i2c_read();
        i@c_read();
      if(state == 0x80)
       i2c_read(2);
      if(state >= 0x80)
         i2c_write(send_buffer[state - 0x80]);
      else if(state > 0)
         rcv_buffer[state - 1] = i2c_read();
}
```

**Example Files:**
ex_slave.c

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

## i2c_poll( )

**Syntax:**
i2c_poll()
i2c_poll(**stream**)

**Parameters:**
**stream** (optional)- specify the stream defined in #USE I2C

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

**Function:**
The **i2c_poll()** function should only be used when the built-in SSP is used. This function returns TRUE if the hardware has a received byte in the buffer. When a TRUE is returned, a call to **i2c_read()** will immediately return the byte that was received.

**Availability:**
Devices with built-in I2C

**Requires:**
#USE I2C

**Examples:**
```
  if(i2c-poll())
  buffer [index]=i2c-read();//read data
  }
```

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

## i2c_read( )

**Syntax:**
data = i2c_read();
data = i2c_read(ack);
data = i2c_read(stream, ack);

**Parameters:**
***ack*** -Optional, defaults to 1
0 indicates do not ack
1 indicates to ack
2 slave only, indicates to not release clock at end of read. Use when **i2c_isr_state()** returns 0x80

**stream** - specify the stream defined in #USE I2C

**Returns:**
data - 8 bit int

**Function:**
Reads a byte over the I2C interface. In master mode this function will generate the clock and in slave mode it will wait for the clock. There is no timeout for the slave, use

**i2c_poll()** to prevent a lockup. Use **restart_wdt()** in the **#USE I2C** to strobe the watch-dog timer in the slave mode while waiting.

**Availability:**
All devices

**Requires:**
#USE I2C

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

**Example Files:**
ex_extee.c **with** 2416.c

**See Also:**
i2c_poll, i2c_speed, i2c_start, i2c_stop, i2c_slaveaddr, i2c_isr_state, i2c_write, #USE I2C, I2C Overview

## i2c_slaveaddr( )

**Syntax:**
i2c_slaveaddr(**addr**);
i2c_slaveaddr(stream, **addr**)

**Parameters:**
**addr** = 8 bit device address

**stream**(optional) - specifies the stream used in #USE I2C

**Returns:**
---

**Function:**
This functions sets the address for the I2C interface in slave mode.

**Availability:**
Devices with built-in I2C

**Requires:**
#USE I2C

**Examples:**
```
i2c_SlaveAddr(0x08);
i2c_SlaveAddr(i2cStream1, 0x08)
```

**Example Files:**
ex_slave.c

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

## i2c_speed( )

**Syntax:**
i2c_speed (***baud***)
i2c_speed **(stream**, ***baud***)

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

**stream** - specify the stream defined in #USE I2C

**Returns:**
---

**Function:**
This function changes the I2c bit rate at run time. This only works if the hardware I2C module is being used.

**Availability:**
All Devices

**Requires:**
#USE I2C

**Examples:**
```
i2C_Speed (400000);
putc(13)
```

**Example Files:**
ex_tgetc.c

**See Also:**
i2c_poll, i2c_start, i2c_stop, i2c_slaveaddr, i2c_isr_state, i2c_write, i2c_read, #USE I2C, I2C Overview

## i2c_start( )

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

**Parameters:**
**stream -** specify the stream defined in #USE I2C
**restart**:- 2 - new restart is forced instead of start
1 - normal start is performed
0 - (or not specified) – restart is done only if the compiler last encountered a **i2c_start()** and no **i2c_stop()**

**Returns:**
Undefined

**Function:**
Issues a start condition when in the I2C master mode. After the start condition the clock is held low until **i2c_write()** is called. If another **i2c_start()** is called in the same function before an **i2c_stop()** is called, then a special restart condition is issued.

Note that specific I2C protocol depends on the slave device. The **i2c_start()** function will now accept an optional parameter. If 1 the compiler assumes the bus is in the stopped state. If 2 the compiler treats this **i2c_start()** as a restart. If no parameter is passed a 2 is used only if the compiler compiled a **i2c_start()** last with no **i2c_stop()** since.

**Availability:**
All Devices

**Requires:**
#USE I2C

**Examples:**
```
i2c_start();
i2c_write(0xa0);     // Device address
i2c_write(address);  // Data to device
i2c_start();         // Restart
i2c_write(0xa1);     // to change data direction
```

```
data=i2c_read(0);     // Now read from slave
i2c_stop()
```

**Example Files:**
ex_extee.c with 2416.c

**See Also:**
i2c_poll, i2c_speed, i2c_stop, i2c_slaveaddr, i2c_isr_state, i2c_write, i2c_read, #USE I2C, I2C Overview

## i2c_stop( )

**Syntax:**
i2c_stop()
i2c_stop(**stream**)

**Parameters:**
**stream -** (optional) specify the stream defined in #USE I2C

**Returns:**
Undefined

**Function:**
Issues a stop condition when in the I2C master mode.

**Availability:**
All Devices

**Requires:**
#USE I2C

**Examples:**
```
i2c_start();          // Start condition
i2c_write(0xa0);      // Device address
i2c_write(5);        // Device command
i2c_write(12);       // Device data
i2c_stop();          // Stop condition
```

**Example Files:**
ex_extee.c with 2416.c

**See Also:**
i2c_poll, i2c_speed, i2c_start, i2c_slaveaddr, i2c_isr_state, i2c_write, i2c_read, #USE I2C, I2C Overview

## i2c_transfer( )

**Syntax:**
i2c_transfer([**stream**], **address, wData, wCount,** [**rData**], [**rCount**]);

**Parameters:**
***stream*** - Optional, the stream defined in #USE I2C to use.

***address*** - The device address to transfer data to and from.

***wData*** - Pointer to data to transfer to device.

***wCount*** - Number of bytes to transfer to device.

***rData*** - Optional, pointer to save transferred data from device to.

***Rcount*** - Optional, number of byte to transfer from device. Must be used if rData is used.

**Returns:**
0 for ACK and 1 for NACK. When only writing data, it returns whether the Slave device ACK'd or NACK'd the write command or last byte transmitted; whichever occurred last. If writing and reading data, if the Slave NACK'd one of the bytes that were transmitted, it returns a NACK, otherwise it returns whether the Slave ACK'd or NACK'd the read command.

**Function:**
Transfer data to and from an I2C device. This function does the I2C start, restart, write, read and stop operations. If the Slave device NACK's the write command, read command or one of the write bytes, the function will exit at that point even if it did not finish writing and/or reading all the data.

**Availability:**
All devices when #USE I2C is setup for Master Mode.

**Requires:**
---

**Examples:**
```
  unsigned int8 rAddress=0;
```

```
unsigned int8 rData[16];
int1 ack;

ack = i2c_transfer(0xA0,&rAddress, 1, rData, 16);

if(ack==0)
     printf("\r\nData transferred successfully");
else
     printf("\r\nData transferred unsuccessfully");
```

**Example File:**
ex_i2c_master_hw_k42.c

**See Also:**
i2c_poll(), i2c_speed(), i2c_stop(), i2c_slaveaddr(), i2c_isr_state(), i2c_write(), i2c_read(), i2c_transfer_out(), i2c_transfer_in(), #USE_I2C, I2C Overview

## i2c_transfer_in( )

**Syntax:**
i2c_transfer_in([**stream**], **address, rData**, **rCount**);

**Parameters:**
***stream*** - Optional, the stream defined in #USE I2C to use.

***address*** - The device address to transfer data from.

***rData*** - Optional, pointer to save transferred data from device to.

***Rcount*** - Number of byte to transfer from device.

**Returns:**
0 for ACK and 1 for NACK from the read command.

**Function:**
Transfer data to and from an I2C device. This function does the I2C start, restart, write, read and stop operations. If the Slave NACK'd the read command, the function will exit without reading any data.

**Availability:**
All devices when #USE I2C is setup for Master Mode.

**Requires:**
---

**Examples:**
```
unsigned int8 rData[16];
int1 ack;

ack=i2c_transfer_in(0xA0,rData,16);

if(ack==0)
     printf("Data read successfully");
else
     printf("data not read");
```

**Example File:**
ex_i2c_master_hw_k42.c

**See Also:**
i2c_poll(), i2c_speed(), i2c_stop(), i2c_slaveaddr(), i2c_isr_state(), i2c_write(), i2c_read(), i2c_transfer_out(), i2c_transfer(), #USE_I2C, I2C Overview

## i2c_transfer_out( )

**Syntax:**
i2c_transfer_out([**stream**], **address, wData**, w**Count**);

**Parameters:**
***stream*** - Optional, the stream defined in #USE I2C to use.

***address*** - The device address to transfer data to.

***wData*** - Pointer to data to transfer to device.

***wcount*** - Number of bytes to transfer to device.

**Returns:**
0 for ACK and 1 for NACK of either the write command or last byte transmitted, whichever occurred last.

**Function:**
Transfer data to and from an I2C device. This function does the I2C start, restart, write, read and stop operations. If the Slave device NACK's the write command or one of the write bytes the function will exit at that point, even if it did not finish writing all the data.

**Availability:**
All devices when #USE I2C is setup for Master Mode.

**Requires:**
---

**Examples:**
```
unsigned int8wData[16];
int1 ack;

ack = i2c_transfer_out(0xA0,wData, 16);

if(ack==0)
     printf("\r\nData transferred successfully");
else
     printf("\r\nData transferred unsuccessfully");
```

**Example File:**
ex_i2c_master_hw_k42.c

**See Also:**
i2c_poll(), i2c_speed(), i2c_stop(), i2c_slaveaddr(), i2c_isr_state(), i2c_write(), i2c_read(), i2c_transfer_in(), i2c_transfer(), #USE_I2C, I2C Overview

## i2c_write( )

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

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

**stream** - specify the stream defined in #USE I2C

**Returns:**
This function returns the ACK Bit.
0 means ACK, 1 means NO ACK, 2 means there was a collision if in Multi_Master Mode.
This does not return an ACK if using i2c in slave mode.

**Function:**
Sends a single byte over the I2C interface. In master mode this function will generate a clock with the data and in slave mode it will wait for the clock from the master. No automatic time-out 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

**See Also:**
i2c_poll, i2c_speed, i2c_start, i2c_stop, i2c_slaveaddr, i2c_isr_state, i2c_read, #USE I2C, I2C Overview

## input( )

**Syntax:**
value = input (***pin***)

**Parameters:**
***Pin*** to read. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43.

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

The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. note that doing I/O with a variable instead of a constant will take much longer time.

**Returns:**
0 (or FALSE) if the pin is low,
1 (or TRUE) if the pin is high

**Function:**
This function returns the state of the indicated pin. The method of I/O is dependent on the last USE *_IO directive. By default with standard I/O before the input is done the data direction is set to input.

**Availability:**
All Devices

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

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

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

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

**Example Files:**
ex_pulse.c

**See Also:**
input_x(), output_low(), output_high(), #USE FIXED_IO, #USE FAST_IO, #USE STANDARD_IO, General Purpose I/O

## input_change_x( )

**Syntax:**
value = input_change_a( );
value = input_change_b( );
value = input_change_c( );
value = input_change_d( );

value = input_change_e( );
value = input_change_f( );
value = input_change_g( );
value = input_change_h( );
value = input_change_j( );
value = input_change_k( );
value = input_change_l( );

**Parameters:**
---

**Returns:**
An 8-bit or 16-bit int representing the changes on the port

**Function:**
This function reads the level of the pins on the port and compares them to the results the last time the **input_change_x( )** function was called. A 1 is returned if the value has changed, 0 if the value is unchanged.

**Availability:**
All Devices

**Requires:**
---

**Examples:**
```
pin_check = input_change_b( );
```

**See Also:**
input( ), input_x( ), output_x( ), #USE FIXED_IO, #USE FAST_IO, #USE STANDARD_IO, General Purpose I/O

## input_state( )

**Syntax:**
value = input_state(***pin***)

**Parameters:**
***pin*** to read. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43.

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

**Returns:**
Bit specifying whether pin is high or low. A 1 indicates the pin is high and a 0 indicates it is low.

**Function:**
This function reads the level of a pin without changing the direction of the pin as INPUT() does.

**Availability:**
All Devices

**Requires:**
---

**Examples:**
```
level = input_state(pin_A3);
printf("level: %d",level)
```

**See Also:**
input(), set_tris_x(), output_low(), output_high(), General Purpose I/O

## input_x( )

**Syntax:**
value = input_a()
value = input_b()
value = input_c()
value = input_d()
value = input_e()
value = input_f()
value = input_g()
value = input_h()
value = input_j()
value = input_k()
value = input_l()

**Parameters:**
---

**Returns:**
An 8 bit int representing the port input data.
*[PCD]* An 16 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.

*[PCD]* Inputs an entire word from a port. The direction register is changed in accordance with the last specified #USE *_IO directive. By default with standard I/O before the input is done the data direction is set to input.

**Availability:**
All Devices

**Requires:**
---

**Examples:**
```
data = input_b();
```

**See Also:**
input(), output_x(), #USE FIXED_IO, #USE FAST_IO, #USE STANDARD_IO

## interrupt_active( )

**Syntax:**
interrupt_active (**interrupt**)

**Parameters:**
**Interrupt** – constant specifying the interrupt

**Returns:**
Boolean value

**Function:**
The function checks the interrupt flag of the specified interrupt and returns true in case the flag is set.

**Availability:**
Devices with Interrupts

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

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

**See Also:**
Interrupts Overview, clear_interrupt, enable_interrupts(), disable_interrupts(), #INT, disable_interrupts() , #INT

## interrupt_enabled()

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

**Syntax:**
interrupt_enabled(**interrupt**);

**Parameters:**
**interrupt-** constant specifying the interrupt

**Returns:**
Boolean value

**Function:**
The function checks the interrupt enable flag of the specified interrupt and returns TRUE when set.

**Availability:**
Devices with Interrupts

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

**Examples:**
```
if(interrupt_enabled(INT_RDA))
    disable_interrupt(INT_RDA);
```

**See Also:**

Interrupts Overview, clear_interrupt, interrupt_active(), disable_interrupts(), #INT, #INT

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

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

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

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

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

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

**Availability:**
All Devices

**Requires:**
#INCLUDE <ctype.h>

**Examples:**
```
  char id[20];
     ...
  if(isalpha(id[0])) {
     valid_id=TRUE;
     for(i=1;i<strlen(id);i++)
      valid_id=valid_id && isalnum(id[i]);
  } else
     valid_id=FALSE;
```

**Example Files:**
ex_str.c

**See Also:**
isamong()

## isamong( )

**Syntax:**
result = isamong (***value, cstring***)

**Parameters:**
**value -** is a character
***cstring* -** is a constant sting

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

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

**Availability:**
All devices

**Requires:**
---

**Examples:**
```
  char x='x';
```

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

**Example Files:**
#INCLUDE <ctype.h>

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

## itoa( )

**Syntax:**
**string** = itoa(**i32value**, **i8base, string**)
*[PCD]* **string** = itoa(**i48value, i8base, string)**
*[PCD]* **string** = itoa(**i64value, i8base, string**)

**Parameters:**
**i32value** is a 32 bit int
*[PCD]* **i48value** is a 48 bit int
*[PCD]* **i64value** is a 64 bit int

**i8base** is a 8 bit int

**string** is a pointer to a null terminated string of characters

**Returns:**
**string** is a pointer to a null terminated string of characters

**Function:**
Converts the signed int32 to a string according to the provided base and returns the converted value if any. If the result cannot be represented, the function will return 0.
*[PCD]* Converts the signed int48, or a int64 to a string according to the provided base and returns the converted value if any. If the result cannot be represented, the function will return 0.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**
```
int32 x=1234;
char string[5];

itoa(x,10, string);      // string is now "1234"
```

## jump_to_isr( )

**Syntax:**
jump_to_isr (***address***)

**Parameters:**
***address*** is a valid program memory address

**Returns:**
---

**Function:**
The jump_to_isr function is used when the location of the interrupt service routines are not at the default location in program memory. When an interrupt occurs, program execution will jump to the default location and then jump to the specified address.

**Availability:**
All Devices

**Requires:**
---

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

**Example Files:**
ex_bootloader.c

**See Also:**
#BUILD

## kbhit( )

**Syntax:**
value = kbhit()

value = kbhit (***stream***)

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

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

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

**Availability:**
All Devices

**Requires:**
#USE RS232

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

**Example Files:**
ex_tgetc.c

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

## label_address( )

**Syntax:**
value = label_address(***label***);

**Parameters:**
***label -*** is a C label anywhere in the function

**Returns:**
16 bit int in PCB and PCM and 32 bit int for PCH
*[PCD]* 32 bit int for PCD

**Function:**
This function obtains the address in ROM of the next instruction after the label. This is not a normally used function except in very special situations.

**Availability:**
All Devices

**Requires:**
---

**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

**See Also:**
goto_address()

## labs( )

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

**Parameters:**
***value is a 16 bit signed long int***
*[PCD]* ***value*** is a 32, 48 or 64 bit signed long int

**Returns:**
A 16 bit signed long int
*[PCD]* A signed long int of type ***value***

**Function:**
Computes the absolute value of a long integer.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

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

**See Also:**
abs()

## lcd_contrast( )

**Syntax:**
lcd_contrast(***contrast***)

**Parameters:**
***contrast*** is used to set the internal contrast control resistance ladder

**Returns:**
Undefined

**Function:**
This function controls the contrast of the LCD segments with a value passed in between 0 and 7. A value of 0 will produce the minimum contrast, 7 will produce the maximum contrast.

**Availability:**
Only on select devices with built-in LCD Driver Module

**Requires:**
---

**Examples:**
```
lcd_contrast( 0 );      // Minimum Contrast
```

```
lcd_contrast( 7 );      // Maximum Contrast
```

**See Also:**
lcd_load( ), lcd_symbol( ), setup_lcd( ), Internal LCD Overview

## 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 to the LCD segment memory.

**Returns:**
Undefined

**Function:**
This function will load ***length*** bytes from ***buffer_pointer*** into the LCD segment memory beginning at ***offset***. The **lcd_symbol( )** function provides as easier way to write data to the segment memory.

**Availability:**
Only on select devices with built-in LCD Driver Module

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

**Examples:**

```
lcd_load(buffer, 0, 16);
```

**Example Files:**
ex_92lcd.c

**See Also:**
lcd_symbol(), setup_lcd(), lcd_contrast( ), Internal LCD Overview

## lcd_symbol( )

**Syntax:**
lcd_symbol (***symbol, bX_addr***);

**Parameters:**
***symbol*** is a 8 bit or 16 bit constant.

***bX_addr*** *is a bit address representing the segment location to be used for bit X of the specified symbol.*
1-16 segments could be specified

**Returns:**
Undefined

**Function:**
This function loads the bits for the symbol into the segment data registers for the LCD with each bit address specified. If bit X in symbol is set, the segment at bX_addr is set, otherwise it is cleared. The bX_addr is a bit address into the LCD RAM.

**Availability:**
Only on select devices with built-in LCD Driver Module

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

**Examples:**
```
byte CONST DIGIT_MAP[10] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE0,
0xFE, 0xE6};

#define DIGIT1   COM1+20, COM1+18, COM2+18, COM3+20, COM2+28, COM1+28,
COM2+20, COM3+18

for(i = 0; i <= 9; i++) {
     lcd_symbol( DIGIT_MAP[i], DIGIT1 );
     delay_ms( 1000 );
}
```

**Example Files:**
ex_92lcd.c

**See Also:**
setup_lcd(), lcd_load(), lcd_contrast( ), Internal LCD Overview

## ldexp( )

**Syntax:**
result= ldexp (***value***, ***exp***);

**Parameters:**
***value*** is float
*[PCD]* ***value*** any float type

***exp*** is a signed int

**Returns:**
Result is a float with value result times 2 raised to power exp.
*[PCD]* Result will have a precision equal to ***value***

**Function:**
The **ldexp()** function multiplies a floating-point number by an integral power of 2.

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

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

**See Also:**
frexp(), exp(), log(), log10(), modf()

## load_slave_program( )

**Syntax:**
load_slave_program(**address, instructions**);

**Parameters:**
***address*** - The address in the Master's program memory that the Slave program is stored at. Because of how the data is stored and written the address must be a multiple of 4.

***Instructions*** - The number of instructions to copy from the Master's Flash to the Slave's PRAM, because of how the data is written the number of instructions must be a multiple of 2.

**Returns:**
---

**Function:**
Copy the Slave program stored in the Master's Flash to the Slave's PRAM.

**Availability:**
Only available on Dual Core devices.

**Requires:**
---

**Examples:**
```
load_slave_program(0x10000, 512);
```

**See Also:**
verify_slave_program()

## log( )

**Syntax:**
result= ldexp (***value***, ***exp***);

**Parameters:**
***value*** is float
*[PCD]* ***value*** any float type

***exp*** is a signed int

**Returns:**
Result is a float with value result times 2 raised to power exp
*[PCD]* Result will have a precision equal to ***value***

**Function:**
Computes the natural logarithm of the float x. If the argument is less than or equal to zero or too large, the behavior is undefined.
Note on error handling: "errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Domain error occurs in the following cases: log: when the argument is negative

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

**Examples:**
```
lnx = log(x);
```

**See Also:**
log10(), exp(), pow()

## log10( )

**Syntax:**
result= log10 (***value***)

**Parameters:**
***value*** is float
*[PCD]* ***value*** any float type

***exp*** is a signed int

**Returns:**
Result is a float with value result times 2 raised to power exp
*[PCD]* Result will have a precision equal to ***value***

**Function:**
Computes the natural logarithm of the float x. If the argument is less than or equal to zero or too large, the behavior is undefined.
Note on error handling: "errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Domain error occurs in the following cases: log10: when the argument is negative

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

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

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

## longjmp( )

**Syntax:**
longjmp (***env, val***)

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

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

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

**Function:**
Performs the non-local transfer of control

**Availability:**
All Devices

**Requires:**
#INCLUDE <setjmp.h>

**Examples:**
```
longjmp(jmpbuf, 1);
```

**See Also:**
setjmp()

## make8( )

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

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

**Returns:**
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:**
---

**Examples:**
```
int32 x;
int y;

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

**See Also:**
make16(), make32()

## make16( )

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

**Parameters:**
***varhigh*** and ***varlow*** are 8 bit integer

**Returns:**
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:**
---

**Examples:**
```
long x;
int hi,lo;

x = make16(hi,lo
```

**Example Files:**
ltc1298.c

**See Also:**
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:**
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:**
---

**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

**See Also:**
make8(), make16()

## malloc( )

**Syntax:**
ptr=malloc(***size***)

**Parameters:**
***size* -** is an integer representing the number of byes to be allocated

**Returns:**
A pointer to the allocated memory, if any. Returns null otherwise

**Function:**
The malloc function allocates space for an object whose size is specified by size and whose value is indeterminate

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlibm.h>

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

**See Also:**
realloc(), free(), calloc()

## memcpy( ) memmove( )

**Syntax:**
memcpy (***destination***, ***source***, ***n***)
memmove(***destination***, ***source***, ***n***)

**Parameters:**
***destination* -** is a pointer to the destination memory
***source* -** is a pointer to the source memory
***n* -** is the number of bytes to transfer

**Returns:**
Undefined

**Function:**
Copies **n** bytes from source to destination in RAM. Be aware that array names are pointers where other variable names and structure names are not (and therefore need a & before them).

**memmove()** performs a safe copy (overlapping objects does not cause a problem). Copying takes place as if the **n** characters from the source are first copied into a temporary array of n characters that does not overlap the destination and source objects. Then the **n** characters from the temporary array are copied to destination.

**Availability:**
All Devices

**Requires:**
---

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

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

**See Also:**
strcpy(), memset()

## memset( )

**Syntax:**
memset (***destination***, ***value***, ***n***)

***P*arameters:**
***destination* -** is a pointer to memory.
***value* -** is a 8 bit int
***n* -** is a 16 bit int
PCB and PCM parts n can only be 1-255.

**Returns:**
Undefined

**Function:**
Sets n number of bytes, starting at destination, to value. Be aware that array names are pointers where other variable names and structure names are not (and therefore need a & before them).

**Availability:**
All Devices

**Requires:**
---

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

**See Also:**
memcpy()

## modf( )

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

**Parameters:**
***value*** is a float
*[PCD]* ***value*** is any float type
***integral*** is a float
*[PCD]* ***integral*** is any float type

**Returns:**
Result is a float
*[PCD]* Result is a float with precision equal to **value**

**Function:**
The **modf()** function breaks the argument value into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a float in the object integral.

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

**Examples:**
```
float result, integral;
result=modf(123.987,&integral);   // result is .987 and integral is
123.000
```

## msi_fifo_status( )

**Syntax:**
Status = smi_fifo_status();

**Parameters:**
---

**Returns:**
An int16 value indicating the status of the MSI FIFO registers. See the device's header file for defines that can be and'ed with return value to determine if the corresponding status bits are set.

**Function:**
Read the status bits from the FIFO Control/Status register from the Master Slave Interface (MSI) peripheral.

**Availability:**
Only available on Dual Core devices.

**Requires:**
---

**Examples:**
```
unsigned int16 Status;
unsigned int16 Value;

Status = msi_fifo_status();

if((Status & MSI_READ_FIFO_EMPTY) == FALSE)
     Value = smi_read_fifo();
```

**See Also:**
msi_write_mailbox(), msi_read_mailbox(), msi_status(), msi_read_fifo(), msi_mailbox_status(), msi_write_fifo(), setup_msi()

## msi_mailbox_status( )

**Syntax:**
Status = msi_mailbox_status( );

**Parameters:**
---

**Returns:**
An int8 value indicating whether the mailbox associated with corresponding handshake protocol has data to be read. Bit 0 is for handshake protocol A, Bit 1 is for handshake protocol B, etc. If the corresponding bit is a 1, then mailbox registers have been written, i.e. has data to be read. If bit is a 0, then mailbox registers have not been written, i.e. has no data to be read, or the corresponding handshake protocol block is disabled.

**Function:**
Read the MSI1MBSX, master core, or SI1MBSX, slave core, register from the Master Slave Interface (MSI) peripheral. This is used to determine if mailboxes associated with the corresponding handshake protocols have data to red or can be written.

**Availability:**
Only available on Dual Core devices.

**Requires:**
---

**Examples:**
```
unsigned int16 Status;

unsigned int16 Value;

Status = msi_mailbox_status():

if(bit_tst(Status, 0) ==1)     //check if protocol A has data to read
     Value = msi_read_mailbox(0);
```

**See Also:**
msi_write_mailbox(), msi_read_mailbox(), msi_status(), msi_read_fifo(), msi_write_fifo(), setup_msi(), msi_fifo_status()

## msi_read_fifo( )

**Syntax:**
Success = msi_read_fifo(***data**);

**Parameters:**
***Data*** - pointer to int16 variable to return read data to.

**Returns:**
True if data was successfully read from FIFO, FALSE is data was not read.

**Function:**
Read from the read FIFO register from the Master Slave Interface (MSI) peripheral. Additionally, the function checks the corresponding Read FIFO Empty Status bit from the FIFO Control/Status Register and only performs the read it, the Status bit indicates that the read FIFO buffer is not empty. This is to keep a read underflow from happening.

**Availability:**
Only available on Dual Core devices.

**Requires:**
---

**Examples:**
```
unsigned int16 Status;
unsigned int16 Value;

Status = msi_read_fifo(&Value):
```

**See Also:**
msi_write_mailbox(), msi_read_mailbox(), msi_status(), msi_fifo_status(), msi_write_fifo(), setup_msi()

## msi_read_mailbox( )

**Syntax:**
Value=msi_read_mailbox(**mailbox**);

**Parameters:**
**mailbox** - The mailbox register to read from can be a value from 0 to 15.

**Returns:**
An int16 value read from the corresponding mailbox register.

**Function:**
Read one of the mailbox registers from the Master Slave Interface (MSI) peripheral. The direction of the mailbox registers depends how the associated configuration bits are set.

This function should only be used to read mailbox registers that are set as read registers for the corresponding core.

**Availability:**
Only available on Dual Core devices.

**Requires:**
---

**Examples:**
```
unsigned int16 Value;

unsigned int8 Mailbox = 0;

Value = msi_read_mailbox(Mailbox);    //reads mailbox 0 register
```

**See Also:**
msi_write_mailbox(), msi_status(), msi_read_fifo(), msi_mailbox_status(), msi_write_fifo(), setup_msi(), msi_fifo_status()

## msi_status( )

**Syntax:**
Status=msi_status( );

**Parameters:**

**Returns:**
The status of the MSI peripheral. See the device's header file for the defines that can be and'd with return value to determine if the corresponding status bits are set.

**Function:**
Read the MSI1STAT, master core, or SI1STAT, slave core, register from the Master Slave Interface (MSI) peripheral.

**Availability:**
Only available on Dual Core devices.

**Requires:**
---

**Examples:**
```
unsigned int16 Status;

Status=msi_status();
```

```
if((Status & MSI_SLAVE_IN_RESET) == MSI_SLAVE_IN_RESET)
     setup_msi(MSI_SLAVE_ENABLE);
```

**See Also:**
msi_write_mailbox(), msi_read_mailbox(), msi_read_fifo(), msi_mailbox_status(), msi_write_fifo(), setup_msi(), msi_fifo_status()

## msi_write_fifo( )

**Syntax:**
Success = msi_write_fifo(**data**);

**Parameters:**
***Data*** - int16 data to write to FIFO register..

**Returns:**
True if data was successfully written to FIFO, FALSE is data was not written..

**Function:**
Write data to the write FIFO register from the Master Slave Interface (MSI) peripheral. Additionally, the function checks the corresponding Write FIFO Empty Status bit from the FIFO Control/Status Register and only performs the write, the Status bit indicates that the write FIFO buffer is not full. This is to keep a write underflow from happening.

**Availability:**
Only available on Dual Core devices.

**Requires:**
---

**Examples:**
```
unsigned int16 Status;
unsigned int16 Value = 0x1122;

Status = msi_write_fifo(Value):
```

**See Also:**
msi_write_mailbox(), msi_read_mailbox(), msi_status(), msi_read_fifo(), msi_fifo_status(), setup_msi()

## msi_write_mailbox( )

**Syntax:**
msi_write_mailbox(**mailbox, data**);

**Parameters:**
**mailbox** - The mailbox register to write from can be a value from 0 to 15.

**data** - An int16 value to write to the mailbox register.

**Returns:**
---

**Function:**
Write to one of the mailbox registers from the Master Slave Interface (MSI) peripheral. The direction of the mailbox registers depends how the associated configuration bits are set. This function should only be used to read mailbox registers that are set as write registers for the corresponding core.

**Availability:**
Only available on Dual Core devices.

**Requires:**
---

**Examples:**
```
unsigned int16 Value = 0x1122;

unsigned int8 Mailbox = 0;

msi_write_mailbox(Mailbox, Value);    //writes mailbox 0 register
```

**See Also:**
msi_read_mailbox(), msi_mailbox_status(), msi_read_fifo(),msi_write_fifo(), msi_fifo_status(), setup_msi()

## mul( )

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

**Parameters:**
***val1*** and ***val2*** are both 8-bit or 16-bit integers
*[PCD]* ***val1*** and ***val2*** are both 8-bit, 16-bit, or 48-bit integers

**Returns:**
A 16-bit integer if both parameters are 8-bit integers, or a 32-bit integer if both parameters are 16-bit integers.
*[PCD]*

| *val1* | *val2* | *prod* |
|---|---|---|
| 8 | 8 | 16 |
| 16* | 16 | 32 |
| 32* | 32 | 64 |
| 48* | 48 | 64** |

* or less
** large numbers will overflow with wrong results

**Function:**
Performs an optimized multiplication. By accepting a different type than it returns, this function avoids the overhead of converting the parameters to a larger type.

**Availability:**
All Devices

**Requires:**
---

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

## nargs( )

**Syntax:**
void foo(char * **str**, int **count**, ...)

**Parameters:**
The function can take variable parameters. The user can use stdarg library to create functions that take variable parameters.

**Returns:**
Function dependent

**Function:**
The stdarg library allows the user to create functions that supports variable arguments.

The function that will accept a variable number of arguments must have at least one actual, known parameters, and it may have more. The number of arguments is often passed to the function in one of its actual parameters. If the variable-length argument list can involve more that one type, the type information is generally passed as well. Before processing can begin, the function creates a special argument pointer of type va_list.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdarg.h>

**Examples:**

```
int foo(int num, ...)
{
 int sum = 0;
 int i;
 va_list argptr;                    // create special argument pointer
 va_start(argptr,num);              // initialize argptr
 for(i=0; i<num; i++)
   sum = sum + va_arg(argptr, int);
 va_end(argptr);                    // end variable processing
 return sum;
}

 void main()
{
 int total;
 total = foo(2,4,6,9,10,2);
}
```

**See Also:**
va_start( ) , va_end( ) , va_arg( )

## offset( ) 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:**
8 bit byte

**Function:**
These functions return an offset into a structure for the indicated field.
**offsetof()** returns the offset in bytes and offsetofbit returns the offset in bits.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stddef.h>

**Examples:**
```
  struct  time_structure {
            int hour, min, sec;
            int zone : 4;
            intl 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 28total =
  foo(2,4,6,9,10,2);
  }
```

## outputx( )

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

**Parameters:**
***value -*** *is a* 8 *bit int*
*[PCD]* ***value -*** *is a* 16 *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.
*[PCD]* Output an entire word to a port. The direction register is changed in accordance with the last specified #USE *_IO directive.

**Availability:**
All Device that include all ports (A-E)

**Requires:**
---

**Examples:**
```
OUTPUT_B(0xf0);
```

**Example Files:**
ex_patg.c

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

## output_bit( )

**Syntax:**
output_bit (***pin***, ***value***)

**Parameters:**
***pins -*** defined in the devices .h file. The actual number is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #DEFINE PIN_A3 43. The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time.
*[PCD]* ***pins -*** defined in the devices .h file. The actual number is a bit address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651. This is defined as follows: #define PIN_A3 5651.

***value*** is a 1 or a 0.

**Returns:**
Undefined

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

**Availability:**
All Devices

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

**Examples:**

```
output_bit(PIN_B0, 0);               // Same as
output_low(pin_B0);
output_bit(PIN_B0,input(PIN_B1));    // Make pin B0 the same as
B1

output_bit(PIN_B0,shift_left(&data,1,input(PIN_B1)));// Output
the MSB of data to
                                     // B0 and at the same time
                                     // shift B1 into the LSB of
data
int16 i=PIN_B0;
ouput_bit(i,shift_left(&data,1,input(PIN_B1)));     //same as
above example, but
                                    //uses a variable instead of
a constant
```

**Example Files:**
ex_extee.c with 9356.c

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

## output_drive( )

**Syntax:**
output_drive(pin)

**Parameters:**
***pins -*** are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #DEFINE PIN_A3 43. The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time.
*[PCD]* ***pins -*** are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651. This is defined as follows: #DEFINE PIN_A3 5651.

**Returns:**
Undefined

**Function:**
Sets the specified pin to the output mode.

**Availability:**
All Devices

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

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

**See Also:**
input(), output_low(), output_high(), output_bit(), output_x(), output_float()

## output_float( )

**Syntax:**
output_float(**pin**)

**Parameters:**
***pins -*** are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #DEFINE PIN_A3 43. The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time.

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

**Returns:**
Undefined

**Function:**
Sets the specified pin to the 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);
```

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

## output_high( )

**Syntax:**
output_high(**pin**)

**Parameters:**
***pin*** to write to. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #DEFINE PIN_A3 43. The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time.
*[PCD]* ***pin*** to write to. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651. This is defined as follows: #DEFINE PIN_A3 5651.

**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);

Int16 i=PIN_A1;
output_low(PIN_A1);
```

**Example Files:**
ex_sqw.c

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

## output_low( )

**Syntax:**
output_low(**pin**)

**Parameters:**
***pin*** to write to. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #DEFINE PIN_A3 43. The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time.
*[PCD]* ***pin*** to write to. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651. This is defined as follows: #DEFINE PIN_A3 5651.

**Returns:**
Undefined

**Function:**
Sets a given pin to the ground state. The method of I/O used is dependent on the last USE *_IO directive.

**Availability:**
All Devices

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

**Examples:**
```
output_low(PIN_A0);

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

**Example Files:**
ex_sqw.c

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

## output_toggle( )

**Syntax:**
output_toggle(**pin**)

**Parameters:**
***pin*** to write to. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #DEFINE PIN_A3 43. The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time.
*[PCD]* ***pin*** to write to. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 0x2C2) bit 3 would have a value of 0x2C2*8+3 or 5651. This is defined as follows: #DEFINE PIN_A3 5651.

**Returns:**
Undefined

**Function:**
Toggles the high/low state of the specified pin.

**Availability:**
All Devices

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

**Examples:**
```
output_toggle(PIN_B4);
```

**See Also:**
input(), output_high(), output_low(), output_bit(), output_x()

## perror( )

**Syntax:**
perror(***string***);

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

**Returns:**
---

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

**Availability:**
All Devices

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

**Examples:**
```
x = sin(y);

if(errno!=0)
```

```
    perror("Problem in find_area");
```

**See Also:**
RS232 I/O Overview

## pid_busy( )

**Syntax:**
result = pid_busy();

**Parameters:**
---

**Returns:**
---

**Function:**
TRUE if PID module is busy or FALSE is PID module is not busy

**Availability:**
All Devices with a PID Module

**Requires:**
---

**Examples:**
```
pid__get_result(PID_START_ONLY, ADCResult);
while(pid_busy());
pid_get_result(PID_READ_ONLY, &PIDResult);
```

**See Also:**
setup_pid(), pid_write(), pid_get_result(), pid_read()

## pid_get_result( )

**Syntax:**
pid_get_result(**set_point, input, &output**);
pid_get_result(**mode, set_point, input**);
pid_get_result(**mode, &output**)
pid_get_result(**mode, set_point, input, &output**);

**Parameters:**
**mode** - constant parameter specifying whether to only start the calculation, only read the result, or start the calculation and read the result. The options are defined in the device's header file as:

- pd_start_read
- pid_read_only
- pid_start_only

**set_point** -a 16-bit variable or constant representing the set point of the control system, the value the input from the control system is compared against to determine the error in the system.

**input** - a 16-bit variable or constant representing the input from the control system.

**output** - a structure that the output of the PID module will be saved to. Either pass the address of the structure as the parameter, or a pointer to the structure as the parameter.

**Returns:**
---

**Function:**
To pass the set point and input from the control system to the PID module, start the PID calculation and get the result of the PID calculation. The PID calculation starts, automatically when the input is written to the PID module's input registers.

**Availability:**
All Devices with a PID Module

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

**Examples:**
```
pid_get_result(SetPoint, ADCResult, &PIDOutput);      //Start
and Read
pid_get_result(PID_START_ONLY, SetPoint, ADCResult);  //Start
Only
pid_get_result(PID_READ_ONLY, &PIDResult);            //Read
Only
```

**See Also:**
setup_pid(), pid_read(), pid_write(), pid_busy()

## pid_read( )

**Syntax:**
pid_read(**register, &output**);

**Parameters:**
**register**- constant specifying which PID registers to read. The registers that can be written are defined in the device's header file as:

- pid_addr_accumulator
- pid_addr_output
- pid_addr_z1
- pid_addr_z2
- pid_addr_k1
- pid_addr_k2
- pid_addr_k3

**output** -a 16-bit variable, 32-bit variable or structure that specified PID registers value will be saved to. The size depends on the registers that are being read. Either pass the address of the variable or structure as the parameter, or a pointer to the variable or structure as the parameter.

**Returns:**
---

**Function:**
To read the current value of the Accumulator, Output, Z1, Z2, Set Point, K1, K2 or K3 PID registers. If the PID is busy with a calculation the function will wait for module to finish calculation before reading the specified register.

**Availability:**
All Devices with a PID Module

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

**Examples:**
```
pid_read(PID_ADDR_Z1, &value_z1);
```

**See Also:**
setup_pid(), pid_write(), pid_get_result(), pid_busy()

## pid_write( )

**Syntax:**
pid_write(**register, &output**);

**Parameters:**
**register**- constant specifying which PID registers to read. The registers that can be written are defined in the device's header file as:

pid_addr_accumulator
pid_addr_output
pid_addr_z1
pid_addr_z2
pid_addr_k1
pid_addr_k2
pid_addr_k3

**output** -a 16-bit variable, 32-bit variable or structure that specified PID registers value will be saved to. The size depends on the registers that are being read. Either pass the address of the variable or structure as the parameter, or a pointer to the variable or structure as the parameter.

**Returns:**
---

**Function:**
To write a new value for the Accumulator, Output, Z1, Z2, Set Point, K1, K2 or K3 PID registers. If the PID is busy with a calculation the function will wait for module to finish the calculation before writing the specified register.

**Availability:**
All Devices with a PID Module

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

**Examples:**
```
pid_write(PID_ADDR_Z1, &value_z1);
```

**See Also:**
setup_pid(), pid_read(), pid_get_result(), pid_busy()

## pin_select( )

**Syntax:**
pin_select(**peripheral_pin**, **pin**, [**unlock**],**[lock**])

**Parameters:**
**peripheral**_pin – a constant string specifying which peripheral pin to map the specified pin to. Refer to #pin_select for all available strings. Using "NULL" for the peripheral_pin parameter will unassign the output peripheral pin that is currently assigned to the pin passed for the pin parameter.

**pin** – the pin to map to the specified peripheral pin. Refer to device's header file for pin defines. If the peripheral_pin parameter is an input, passing FALSE for the pin parameter will unassign the pin that is currently assigned to that peripheral pin.

**unlock** – optional parameter specifying whether to perform an unlock sequence before writing the RPINRx or RPORx register register determined by peripheral_pin and pin options. Default is TRUE if not specified. The unlock sequence must be performed to allow writes to the RPINRx and RPORx registers. This option allows calling pin_select() multiple times without performing an unlock sequence each time.

**lock** – optional parameter specifying whether to perform a lock sequence after writing the RPINRx or RPORx registers. Default is TRUE if not specified. Although not necessary it is a good idea to lock the RPINRx and RPORx registers from writes after all pins have been mapped. This option allows calling pin_select() multiple times without performing a lock sequence each time.

**Returns:**
---

**Availability:**
On device with remappable peripheral pins.

**Requires:**
Pin defines in device's header file.

**Examples:**
```
pin_select("U2TX",PIN_B0);            //Maps PIN_B0 to U2TX
peripheral pin,
                                      //performs unlock and lock
sequences.

pin_select("U2TX",PIN_B0,TRUE,FALSE); //Maps PIN_B0 to U2TX
peripheral pin
```

```
                                        //and performs unlock
  sequence.

  pin_select("U2RX",PIN_B1,FALSE,TRUE); //Maps PIN_B1 to U2RX
  peripheral pin
                                        //and performs lock
  sequence.
```

**See Also:**
#pin_select

## pll_locked( )

**Syntax:**
result=pll_locked();

**Parameters:**
**-----**

**Returns:**
A short int.
TRUE if the PLL is locked/ready,
FALSE if PLL is not locked/ready

**Function:**
Allows testing the PLL Ready Flag bit to determined if the PLL is stable and running.

**Availability:**
All Devices with a Phase Locked Loop (PLL).
Not all devices have a PLL Ready Flag, for those devices the pll_locked() function will always return TRUE

**Requires:**
-----

**Examples:**

```
  while(!pll_locked())
```

**See Also:**
#use delay

## pmp_address(address )

**Syntax:**
pmp_address ( ***address*** );

**Parameters:**
***address***- The address which is a 16 bit destination address value. This will setup the address register on the PMP module and is only used in Master mode.

**Returns:**
Undefined

**Function:**
Configures the address register of the PMP module with the destination address during Master mode operation. The address can be either 14, 15 or 16 bits based on the multiplexing used for the Chip Select Lines 1 and 2.

**Availability:**
All Devices with a built-in Parallel Port Module

**Requires:**
---

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

**See Also:**
setup_pmp(), pmp_address(), pmp_read(), psp_read(), psp_write(), pmp_write(), psp_output_full(), psp_input_full(), psp_overflow(), pmp_output_full(), pmp_input_full(),pmp_overflow()

## pmp_output_full( ) pmp_input_full( ) pmp_overflow( ) pmp_error( ) pmp_timeout( )

**Syntax:**
result = pmp_output_full()
result = pmp_input_full()
result = pmp_overflow()
result = pmp_eror( )
result = pmp_timeout( )

**Parameters:**
---

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

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

**Availability:**
Only available on devices with Parallel Port

**Requires:**
---

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

**See Also:**
setup_pmp(), pmp_write(), pmp_read()

## pmp_read( )

**Syntax:**
result = pmp_read ( );
result = pmp_read8(**address**);
result = pmp_read16(**address**);
pmp_read8(**address**,pointer,count);
pmp_read16(**address,pointer**,**count**);

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

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

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

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

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

**Availability:**
Only available on devices with Parallel Port or an Enhanced Parallel Master Port module.

**Requires:**
---

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

**See Also:**
setup_pmp(), setup_pmp_csx(), pmp_address(), pmp_read(), psp_read(), psp_write(), pmp_write(), psp_output_full(), psp_input_full(), psp_overflow(), pmp_output_full(), pmp_input_full(),pmp_overflow()

## pmp_write( )

**Syntax:**
pmp_write (**data**);

pmp_write8(**address,data**);
pmp_write8(**address,pointer,data**);
pmp_write16(**address,data**);
pmp_write16(**address,pointer,data**);

**Parameters:**
**data**- The byte of data to be written.

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

**pointer**- EPMP only, pointer to data to be written

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

**Returns:**
Undefined

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

**Availability:**
Only available on devices with Parallel Port or an Enhanced Parallel Master Port module.

**Requires:**
---

**Examples:**

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

**See Also:**
setup_pmp(), setup_pmp_csx(), pmp_address(), pmp_read(), psp_read(), psp_write(), pmp_write(), psp_output_full(), psp_input_full(), psp_overflow(), pmp_output_full(), pmp_input_full(), pmp_overflow()

## port_a_current_source( )

**Syntax:**
port_a_current_source(***mask***);

**Parameters:**
***mask* -** an int8 value indicating which port pins have the weak current source enabled. 1 indicates the weak current source is enabled.

**Returns:**
---

**Function:**
Used to enable and disable the weak current source on port A pins.

**Availability:**
Devices that have a weak current source on some of the port A pins.

**Requires:**
---

**Examples:**
```
port_a_current_source(0x0C);      //enables weak current source
                                  //on PIN_A2 and PIN_A3.
```

**See Also:**
set_tris_x( ), get_trisx( ), output_x( ), input_x( ), input_change_x( ), port_x_pullups( ), input( ),
input_state( ), output_low( ), output_high( ), output_toggle( ), output_bit( ), output_float( ), output_drive( ), General Purpose I/O

## port_x_pullups( )

**Syntax:**
port_a_pullups (***value***)
port_b_pullups (***value***)
port_d_pullups (***value***)
port_e_pullups (***value***)

port_j_pullups (***value***)
port_k_pullups (***value***)
port_l_pullups (***value***)
port_x_pullups (***upmask***)
port_x_pullups (***upmask***, ***downmask***)

**Parameters:**
***value -*** is TRUE or FALSE on most parts, some parts that allow pullups to be specified on individual pins permit an 8 bit int here, one bit for each port pin.

***upmask -*** for ports that permit pullups to be specified on a pin basis. This mask indicates what pins should have pullups activated. A 1 indicates the pullups is on.

***downmask -*** for ports that permit pulldowns to be specified on a pin basis. This mask indicates what pins should have pulldowns activated. A 1 indicates the pulldowns is on.

**Returns:**
Undefined

**Function:**
Sets the input pullups. TRUE will activate, and a FALSE will deactivate.

**Availability:**
Only 14 and 16 bit devices (PCM and PCH). (Note: use SETUP_COUNTERS on PCB parts).

**Requires:**
---

**Examples:**
```
port_a_pullups(FALSE);
```

**Example Files:**
ex_lcdkb.c, kbd.c

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

## pow( ) pwr( )

**Syntax:**
f = pow (***x***,***y***)
f = pwr (***x***,***y***)

**Parameters:**
***x*** and ***y*** are of type float
*[PCD]* ***x*** and **y** are any float type

**Returns:**
A float
*[PCD]* A float with precision equal to function parameters x and y.

**Function:**
Calculates X to the Y power.

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

Range error occurs in the following case: pow: when the argument X is negative

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

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

## prgx_status()

**Syntax:**
status = prg1_status();
status = prg2_status();
status = prg3_status();
status = prg4_status();

**Parameters:**
---
**Returns:**
An 8-bit value indicating the status of the PRGx module. See the device's header file for constants that can be returned by function.

**Function:**
Used to set the PRGx modules.

**Availability:**
Devices that have a Programmable Ramp Generator (PRG) module.

**Requires:**
---

**Examples:**

```
int8 Status;
Status = prg1_status();
```

**See Also:**
setup_prgx( )

## printf( ) fprintf( )

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

**Parameters:**
***String*** is a constant string or an array of characters null terminated.

***C String*** is a constant string. Note that format specifiers cannot be used in RAM strings.

***Values*** is a list of variables separated by commas, fname is a function name to be used for outputting (default is **putc** is none is specified.

***Stream*** is a stream identifier (a constant byte)

**Returns:**
Undefined

**Function:**
Outputs a string of characters to either the standard RS-232 pins (first two forms) or to a specified function. Formatting is in accordance with the string argument. When variables are used this string must be a constant. The % character is used within the string to indicate a variable value is to be formatted and output. Longs in the printf may be 16 or 32 bit. A %% will output a single %. Formatting rules for the % follows.

See the Expressions > Constants and Trigraph sections of this manual for other escape character that may be part of the string.

If **fprintf()** is used then the specified stream is used where **printf()** defaults to STDOUT (the last USE RS232).

Format:
The format takes the generic form %nt. n is optional and may be 1-9 to specify how many characters are to be outputted, or 01-09 to indicate leading zeros, or 1.1 to 9.9 for floating point and %w output. t is the type and may be one of the following:

**c** -- string or character
**u** -- unsigned
**d** -- signed int
**Lu --** long unsigned int
**Ld --** long signed int
**x --** hex int (lower case)
**X --** hex int (upper case
**Lx** -- hex long int (lower case)
**LX** -- hex long int (upper case)
**f --** float with truncated decimal
**g --** float with rounded decimal
**e --** float in exponential format
**w --** unsigned int with decimal place inserted. Specify two numbers for ***n***. The first is a total field width. The second is the desired number of decimal places.

Example Formats:

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

* Result is undefined - Assume garbage.

**Availability:**
All Devices

**Requires:**
#USE RS232 (unless fname is used)

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

**Example Files:**
ex_admm.c, ex_lcdkb.c

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

## profileout( )

**Syntax:**
profileout(**string**);
profileout(**string, value**);
profileout(**value**);

**Parameters:**
**string -** is any constant string, and value can be any constant or variable integer. Despite the length of string the user specifies here, the code profile run-time will actually only send a one or two byte identifier tag to the code profile tool to keep transmission and execution time to a minimum.

**Returns:**
Undefined

**Function:**
Typically the code profiler will log and display function entry and exits, to show the call sequence and profile the execution time of the functions. By using ***profileout()***, the user can add any message or display any variable in the code profile tool. Most messages sent by ***profileout()*** are displayed in the 'Data Messages' and 'Call Sequence' screens of the code profile tool.

If a ***profileout(string)*** is used and the first word of string is "START", the code profile tool will then measure the time it takes until it sees the same ***profileout(string)*** where the "START" is replaced with "STOP". This measurement is then displayed in the 'Statistics' screen of the code profile tool, using string as the name (without "START" or "STOP")

**Availability:**
All Devices

**Requires:**
**#use profile()** used somewhere in the project source code

**Examples:**

```
                                        // send a simple string.
 profileout("This is a text string"); // send a variable with a
 string identifier.
 profileout("RemoteSensor=", adc);     // just send a variable.
 profileout(adc);                      // time how long a block of
 code takes to execute.
                                        // this will be displayed
 in the 'Statistics'
                                        // of the Code Profile
 tool.
 profileout("start my algorithm");
  /* code goes here */
 profileout("stop my algorithm")
```

**Example Files:**
ex_profile.c

**See Also:**
#use profile(), #profile, Code Profile Overview

## psmc_blanking( )

**Syntax:**
psmc_blanking(***unit, rising_edge, rise_time, falling_edge, fall_time***);

**Parameters:**
***unit* -** is the PSMC unit number 1-4

***rising_edge* -** are the events that are ignored after the signal activates.

***rise_time* -** is the time in ticks (0-255) that the above events are ignored.

***falling_edge* -** are the events that are ignored after the signal goes inactive.

***fall_time* -** is the time in ticks (0-255) that the above events are ignored.
Events:

psmc_event_c1out
psmc_event_c2out
psmc_event_c3out
psmc_event_c4out
psmc_event_in_pin

**Returns:**
Undefined

**Function:**
This function is used when system noise can cause an incorrect trigger from one of the specified events. This function allows for ignoring these events for a period of time around either edge of the signal. See **setup_psmc()** for a definition of a tick.

Pass a 0 or FALSE for the events to disable blanking for an edge.

**Availability:**
All Devices with PSMC module

**Requires:**
---

## psmc_deadband( )

**Syntax:**
psmc_deadband(***unit,rising_edge, falling_edge***);

**Parameters:**
***unit -*** is the PSMC unit number 1-4

***rising_edge -*** is the deadband time in ticks after the signal goes active. If this function is not called, 0 is used.

***falling_edge -*** is the deadband time in ticks after the signal goes inactive. If this function is not called, 0 is used.

**Returns:**
Undefined

**Function:**
This function sets the deadband time values. Deadbands are a gap in time where both sides of a complementary signal are forced to be inactive. The time values are in ticks. See setup_psmc() for a definition of a tick.

**Availability:**
All Devices with PSMC module

**Requires:**
---

**Examples:**
```
                                // 5 tick deadband when the signal
  goes active.
     psmc_deadband(1, 5, 0)
```

**See Also:**
setup_psmc(), psmc_sync(), psmc_blanking(), psmc_modulation(), psmc_shutdown(), psmc_duty(), psmc_freq_adjust(), psmc_pins()

## psmc_duty( )

**Syntax:**
psmc_duty(***unit, pins_used, pins_active_low***);

**Parameters:**
***unit -*** is the PSMC unit number 1-4

***fall_time -*** is the time in ticks that the signal goes inactive (after the start of the period) assuming the event PSMC_EVENT_TIME has been specified in the setup_psmc().

**Returns:**
Undefined

**Function:**
This function changes the fall time (within the period) for the active signal. This can be used to change the duty of the active pulse. Note that the time is NOT a percentage nor is it the time the signal is active. It is the time from the start of the period that the signal will go inactive. If the rise_time was set to 0, then this time is the total time the signal will be active.

**Availability:**
All Devices with PSMC module

**Requires:**
---

**Examples:**

```
                              // For a 10khz PWM, based on Fosc
 divided by 1
                              // the following sets the duty from
                              // 0% to 100% baed on the ADC reading
 while(TRUE)  {
    psmc_duty(1,(read_adc()*(int16)10)/25)*
       (getenv("CLOCK")/1000000));
 }
```

**See Also:**
setup_psmc(), psmc_deadband(), psmc_sync(), psmc_blanking(), psmc_modulation(), psmc_shutdown(), psmc_freq_adjust(), psmc_pins()

## psmc_freq_adjust( )

**Syntax:**
psmc_freq_adjust(***unit, freq_adjust***);

**Parameters:**
***unit -*** is the PSMC unit number 1-4

***freq_adjust -*** is the time in tick/16 increments to add to the period. The value may be 0-15.

**Returns:**
Undefined

**Function:**
This function adds a fraction of a tick to the period time for some modes of operation.

**Availability:**
All Devices with PSMC module

**Requires:**
---

**See Also:**
setup_psmc(), psmc_deadband(), psmc_sync(), psmc_blanking(), psmc_modulation(), psmc_shutdown(), psmc_dutyt(), psmc_pins()

## psmc_modulation( )

**Syntax:**
psmc_modulation(***unit, options***);

**Parameters:**
***unit*** is the PSMC unit number 1-4

***Options*** may be one of the following:
- psmc_mod_off
- psmc_mod_active
- psmc_mod_inactive
- psmc_mod_c1out
- psmc_mod_c2out
- psmc_mod_c3out
- psmc_mod_c4out
- psmc_mod_ccp1
- psmc_mod_ccp2
- psmc_mod_in_pin

The following may be OR'ed with the above
- psmc_mod_invert
- psmc_mod_not_bdf
- psmc_mod_not_ace

**Returns:**
Undefined

**Function:**
This function allows some source to control if the PWM is running or not. The active/inactive are used for software to control the modulation. The other sources are hardware controlled modulation. There are also options to invert the inputs, and to ignore some of the PWM outputs for the purpose of modulation.

**Availability:**
All Devices with PSMC module

**Requires:**
---

**See Also:**
setup_psmc(), psmc_deadband(), psmc_sync(), psmc_blanking(), psmc_shutdown(), psmc_duty(), psmc_freq_adjust(), psmc_pins()

## psmc_pins( )

**Syntax:**
psmc_pins(***unit, pins_used, pins_active_low***);

**Parameters:**
***unit -*** is the PSMC unit number 1-4

***used_pins -*** is the any combination of the following or'ed together:
psmc_A
psmc_B
psmc_C
psmc_D
psmc_E
psmc_F
psmc_on_next_period

If the last constant is used, all the changes made take effect on the next period (as opposed to immediate)

***pins_active_low -*** is an optional parameter. When used it lists the same pins from above as the pins that should have an inverted polarity.

**Returns:**
Undefined

**Function:**
This function identified the pins allocated to the PSMC unit, the polarity of those pins and it enables the PSMC unit. The tri-state register for each pin is set to the output state.

**Availability:**
All Devices with PSMC module

**Requires:**
---

**Examples:**

```
                              // Simple PWM, 10khz out on pin C0
 assuming a 20mhz crystal
                              // Duty is initially set to 25%
 setup_psmc(1, PSMC)SINGLE,
   PSMC_EVENT_TIME | PSMC_SOURCE_FOSC, us(100,
   PSMC_EVENT_TIME, 0,
   PSMC_EVENT_TIME, us(25));
```

```
psmc_pins(1, PSMC_A);
}
```

**See Also:**
setup_psmc(), psmc_deadband(), psmc_sync(), psmc_blanking(), psmc_modulation(), psmc_shutdown(), psmc_duty(), psmc_freq_adjust()

## psmc_shutdown( )

**Syntax:**
psmc_shutdown(***unit, options, source, pins_high***);
psmc_shutdown(***unit, command***);

**Parameters:**
***unit -*** is the PSMC unit number 1-4

***Options*** may be one of the following:
psmc_shutdown_
psmc_shutdown_normal
psmc_shutdown_auto_restart

***command*** may be one of the following:
psmc_shutdown_restart
psmc_shutdown_force
psmc_shutdown_check

***source*** may be any of the following or'ed together:
psmc_shutdown_c1out
psmc_shutdown_c2out
psmc_shutdown_c3out
psmc_shutdown_c4out
psmc_shutdown_in_pin

***pins_high*** is any combination of the following or'ed together:
psmc_A
psmc_B
psmc_C
psmc_D
psmc_E
psmc_F

**Returns:**
Non-zero if the unit is now in shutdown

**Function:**
This function implements a shutdown capability. When any of the listed events activate the PSMC unit will shutdown and the output pins are driver low unless they are listed in the pins that will be driven high.

The auto restart option will restart when the condition goes inactive, otherwise a call with the restart command must be used. Software can force a shutdown with the force command.

**Availability:**
All Devices with PSMC module

**Requires:**
---

**See Also:**
setup_psmc(), psmc_deadband(), psmc_sync(), psmc_blanking(), psmc_modulation(), psmc_duty(), psmc_freq_adjust(), psmc_pins()

## psmc_sync( )

**Syntax:**
psmc_sync(***slave_unit, master_unit, options***);

**Parameters:**
***slave_unit*** is the PSMC unit number 1-4 to be controlled.

***master_unit*** is the PSMC unit number 1-4 to be synchronized to

Options may be:
- psmc_source_is_phase
- psmc_source_is_period
- psmc_source_disconnect

The following may be OR'ed with the above:
- psmc_invert_duty
- psmc_invert_period

**Returns:**
Non-zero if the unit is now in shutdown

**Function:**
This function allows one PSMC unit (the slave) to be synchronized (the outputs) with another PSMC unit (the master).

**Availability:**
All Devices with PSMC module

**Requires:**
---

**See Also:**
setup_psmc(), psmc_deadband(), psmc_sync(), psmc_modulation(), psmc_shutdown(), psmc_duty(), psmc_freq_adjust(), psmc_pins()

## psp_output_full( ) psp_input_full( ) psp_overflow( ) psp_error( ) psp_timeout( )

**Syntax:**
result = psp_output_full()
result = psp_input_full()
result = psp_overflow()
result = psp_error();
result = psp_timeout();

**Parameters:**
---

**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:**
All Devices with PSP module

**Requires:**
---

**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

**See Also:**
setup_psp(), PSP Overview

## psp_read( )

**Syntax:**
Result = psp_read ( );
Result = psp_read (***address***);

**Parameters:**
***address*** - The address of the buffer location that needs to be read. If address is not specified, use the function **psp_read()** which will read the next buffer location.

**Returns:**
A byte of data

**Function:**
***psp_read()*** will read a byte of data from the next buffer location and *psp_read(**address**)* will read the buffer location ***address***.

**Availability:**
Only the devices with a built in Parallel Master Port module of Enhanced Parallel Master Port module

**Requires:**
---

**Examples:**
```
Result = psp_read();     // Reads next byte of data
Result = psp_read(3);    // Reads the buffer location 3
```

**See Also:**
setup_pmp(), pmp_address(), pmp_read(), psp_read(), psp_write(), pmp_write(), psp_output_full(), psp_input_full(), psp_overflow(), pmp_output_full(), pmp_input_full(),pmp_overflow()

## psp_write

**Syntax:**
psp_write (***data***);
psp_write(***address***, ***data***);

**Parameters:**
***address*** - The buffer location that needs to be written to
***data*** - The byte of data to be written

**Returns:**
Undefined

**Function:**
This will write a byte of data to the next buffer location or will write a byte to the specified buffer location.

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

**Requires:**
---

**Examples:**
```
psp_write( data );  // Write the data byte to the next buffer
location
```

**See Also:**
setup_pmp(), pmp_address(), pmp_read(), psp_read(), psp_write(), pmp_write(), psp_output_full(), psp_input_full(), psp_overflow(), pmp_output_full(), pmp_input_full(),pmp_overflow().

## putc_send( ) fputc_send( )

**Syntax:**
putc_send();
fputc_send(**stream**);

**Parameters:**
**stream –** ***parameter specifying the stream defined in #USE RS232***

**Returns:**
Undefined

**Function:**
Function used to transmit bytes loaded in transmit buffer over RS232. Depending on the options used in #USE RS232 controls if function is available and how it works.

If using hardware UARTx with NOTXISR option it will check if currently transmitting. If not transmitting it will then check for data in transmit buffer. If there is data in transmit buffer it will load next byte from transmit buffer into the hardware TX buffer, unless using CTS flow control option. In that case it will first check to see if CTS line is at its active state before loading next byte from transmit buffer into the hardware TX buffer.

If using hardware UARTx with TXISR option, function only available if using CTS flow control option, it will test to see if the TBEx interrupt is enabled. If not enabled it will then test for data in transmit buffer to send. If there is data to send it will then test the CTS flow control line and if at its active state it will enable the TBEx interrupt. When using the TXISR mode the TBEx interrupt takes care off moving data from the transmit buffer into the hardware TX buffer.

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

**Availability:**
All Devices

**Requires:**
#USE RS232

**Examples:**
```
#USE RS232(UART1,BAUD=9600,TRANSMIT_BUFFER=50,NOTXISR)
printf("Testing Transmit Buffer");
while(TRUE){
   putc_send();
}
```

**See Also:**
_USE_RS232( ), rcv_buffer_full( ), tx_buffer_full( ), tx_buffer_bytes( ), getc( ), putc( ) rintf(), setup_uart( ),putc( )

## pwm_off( )

**Syntax:**
pwm_on([stream]);

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

**Returns:**
Undefined

**Function:**
To turn off the PWM signal.

**Availability:**
All Devices

**Requires:**
#USE PWM

**Examples:**
```
#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
 while(TRUE){
   if(kbhit()){
      c = getc();
      if(c=='O')
         pwm_on();
   }
}
```

**See Also:**
#use_pwm, pwm_off(), pwm_set_duty_percent(), pwm_set_duty(), pwm_set_frequency()

## pwm_set_duty( )

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

**Parameters:**
**stream** – optional parameter specifying the stream defined in #USE PWM.
**duty** – an int16 constant or variable specifying the new PWM high time

**Returns:**
Undefined

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

**Availability:**
All Devices

**Requires:**
#USE PWM

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

**See Also:**
#use_pwm, pwm_on(), pwm_off(), pwm_set_frequency(), pwm_set_duty_percent()

## pwm_set_duty_percent( )

**Syntax:**
pwm_set_duty_percent(**[stream], percent**);

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

**percent**- an int16 constant or variable ranging from 0 to 1000 specifying the new PWM duty cycle, D is 0% and 1000 is 100.0%.

**Returns:**
Undefined

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

**Availability:**
All Devices

**Requires:**
#USE PWM

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

**See Also:**
#use_pwm, pwm_on(), pwm_off(), pwm_set_frequency(), pwm_set_duty()

## pwm_set_frequency( )

**Syntax:**
pwm_set_set_frequency([stream],frequency);

**Parameters:**
**stream** – optional parameter specifying the stream defined in #USE PWM.

**frequency** – an int32 constant or variable specifying the new PWM frequency.

**Returns:**
Undefined

**Function:**
To change the frequency of the PWM signal. Warning this may change the resolution of the PWM signal

**Availability:**
All Devices

**Requires:**
#USE PWM

**Examples:**
```
#USE PWM(OUTPUT=PIN_C2, FREQUENCY=10kHz, DUTY=25)
pwm_set_frequency(1000);   //set PWM frequency to 1kHz
```

**See Also:**
#use_pwm, pwm_on(), pwm_off(), pwm_set_duty_percent, pwm_set_duty()

## pwm1_interrupt_active( ) pwm2_interrupt_active( ) pwm3_interrupt_active( ) pwm4_interrupt_active( ) pwm5_interrupt_active( ) pwm6_interrupt_active( )

**Syntax:**
result_pwm1_interrupt_active (***interrupt***)
result_pwm2_interrupt_active (***interrupt***)
result_pwm3_interrupt_active (***interrupt***)

result_pwm4_interrupt_active (***interrupt***)
result_pwm5_interrupt_active (***interrupt***)
result_pwm6_interrupt_active (***interrupt***)

**Parameters:**
***interrupt*** - 8-bit constant or variable. Constants are defined in the device's header file as:

pwm_period_interrupt
pwm_duty_interrupt
pwm_phase_interrupt
pwm_offset_interrupt

**Returns:**
TRUE if interrupt is active. FALSE if interrupt is not active.

**Function:**
Tests to see if one of the above PWM interrupts is active, interrupt flag is set.

**Availability:**
Devices with a 16-bit PWM module

**Requires:**
#USE PWM

**Examples:**
```
if(pwm1_interrupt_active(PWM_PERIOD_INTERRUPT))
   clear_pwm1_interrupt(PWM_PERIOD_INTERRUPT)
```

**See Also:**
setup_pwm(), set_pwm_duty(), set_pwm_phase(), set_pwm_period(), set_pwm_offset(), enable_pwm_interrupt(), clear_pwm_interrupt(), disable_pwm_interrupt()

## [PCD] qei_get_capture()

**Syntax:**
value = qei_get_capture();
value = qei_get_capture(**unit**);

**Parameters:**
**unit** - optional parameter specifying the QEI unit to read the capture value from. Defaults to 1 if not specified.

**Returns:**
The 32-bit capture value of the specified QEI unit.

**Function:**
Used to get the capture value for the specified QEI unit.

**Availability:**
Some devices that have a QEI module. See the device's header file to determine if function is available.

**Requires:**
---

**Examples:**

```
Unsigned int32 Value;
Value = qei_get_capture(1);
```

**See Also:**
setup_qei( ), qei_set_count( ), qei_status( ), qei_set_index_count(), qei_get_index_count(), qei_get_velocity_count(), qei_get_interval_count()

## qei_get_count( )

**Syntax:**
value = qei_get_count( [***type***] );
*[PCD]* value = qei_get_count( [***unit***] );

**Parameters:**
***type*** - Optional parameter to specify which counter to get, defaults to position counter. Defined in devices .h file as:
qei_get_position_count
qei_get_velicity_count

*[PCD]* ***value***- The 16-bit value of the position counter.

*[PCD]* ***unit***- Optional unit number, defaults to 1.

**Returns:**
The 16-bit value of the position counter or velocity counter.
*[PCD]* void

**Function:**
Reads the current 16-bit value of the position or velocity counter.
*[PCD]* Reads the current 16-bit value of the position counter.

**Availability:**
Devices that have the QEI module

**Requires:**
---

**Examples:**
```
value = qei_get_counter(QEI_GET_POSITION_COUNT);
value = qei_get_counter();
value = qei_get_counter(QEI_GET_VELOCITY_COUNT);
[PCD] value = qei_get_counter();
```

**See Also:**
setup_qei() , qei_set_count() , qei_status()

## [PCD] qei_get_index_count()

**Syntax:**
value = qei_get_index_count();
value = qei_get_index_count(**unit**);

**Parameters:**
**unit** - optional parameter specifying the QEI unit to read the index count value from. Defaults to 1 if not specified.

**Returns:**
The 32-bit index count for the specified QEI unit.

**Function:**
Used to get the index count for the specified QEI unit.

**Availability:**
Some devices that have a QEI module. See the device's header file to determine if function is available.

**Requires:**
------

**Examples:**

```
Unsigned int32 IndexCount;
IndexCount = qei_get_index_count(1);
```

**See Also:**
setup_qei( ), qei_set_count( ), qei_status( ), qei_set_index_count(), qei_get_capture(), qei_get_velocity_count(), qei_get_interval_count()

## [PCD] qei_get_interval_count()

**Syntax:**
value = qei_get_interval_count();
value = qei_get_interval_count(**unit**);

**Parameters:**
**unit** - optional parameter specifying the QEI unit to read the interval count value from. Defaults to 1 if not specified.

**Returns:**
The 32-bit interval count for the specified QEI unit.

**Function:**
Used to get the interval count for the specified QEI unit.

**Availability:**
Some devices that have a QEI module. See the device's header file to determine if function is available.

**Requires:**
---

**Examples:**

```
Unsigned int32 IntervalCount;
IntervalCount = qei_get_interval_count(1);
```

**See Also:**
setup_qei( ), qei_set_count( ), qei_status( ), qei_set_index_count(), qei_get_capture(), qei_get_velocity_count(), qei_get_index_count()

## [PCD] qei_get_velocity_count()

**Syntax:**
value = qei_get_velocity_count();
value = qei_get_velocity_count(**unit**);

**Parameters:**
**unit** - optional parameter specifying the QEI unit to read the velocity count value from. Defaults to 1 if not specified.

**Returns:**
The 16-bit velocity count for the specified QEI unit.

**Function:**
Used to get the velocity count for the specified QEI unit.

**Availability:**
Some devices that have a QEI module. See the device's header file to determine if function is available.

**Requires:**
---

**Examples:**

```
Unsigned int32 VelocityCount;
VelocityCount = qei_get_velocity_count(1);
```

**See Also:**
setup_qei( ), qei_set_count( ), qei_status( ), qei_set_index_count(), qei_get_capture(), qei_get_interval_count(), qei_get_index_count()

## qei_set_count( )

**Syntax:**
qei_set_count( ***value*** );
*[PCD]* qei_set_count( [***unit,***] ***value*** )

**Parameters:**
***value*** - The 16-bit value of the position counter.
*[PCD]* ***value*** - The 16-bit value of the position counter.

*[PCD]* ***unit***- Optional unit number, defaults to 1.

**Returns:**
Void

**Function:**
Write a 16-bit value to the position counter.

**Availability:**
Devices that have the QEI module

**Requires:**
---

**Examples:**
```
  qei_set_counter(value);
```

**See Also:**
setup_qei() , qei_get_count() , qei_status()

## [PCD] qei_set_index_count()

**Syntax:**
qei_set_index_count();
qei_set_index_count(**unit, count**);

**Parameters:**
**unit** - optional parameter specifying the QEI unit to set the index count value from. Defaults to 1 if not specified.
**count** - the 32-bit value to set the index count to.

**Returns:**
---

**Function:**
Used to set the index count for the specified QEI unit.

**Availability:**
Some devices that have a QEI module. See the device's header file to determine if function is available.

**Requires:**
------

**Examples:**

```
  qei_set_index_count(1, 500);
```

**See Also:**
setup_qei( ), qei_set_count( ), qei_status( ), qei_get_velocity_count(), qei_get_capture(), qei_get_interval_count(), qei_get_index_count()

## qei_status( )

**Syntax:**
status = qei_status( );
*[PCD]* status = qei_status( [***unit***] );

**Parameters:**
None
*[PCD]* ***status***- The status of the QEI module

*[PCD]* ***unit***- Optional unit number, defaults to 1

**Returns:**
The status of the QEI module.
*[PCD]* Void

**Function:**
Returns the status of the QEI module.
*[PCD]* Returns the status of the QUI module

**Availability:**
Devices that have the QEI module

**Requires:**
---

**Examples:**
```
status = qei_status();
```

**See Also:**
setup_qei() , qei_set_count() , qei_get_count()

## qsort( )

**Syntax:**
qsort (***base, num, width, compare***)

**Parameters:**
**base**: Pointer to array of sort data

**num**: Number of elements

**width**: Width of elements

**compare**: Function that compares two elements

**Returns:**
---

**Function:**
Performs the shell-metzner sort (not the quick sort algorithm). The contents of the array are sorted into ascending order according to a comparison function pointed to by compare

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**
```
int nums[5]={ 2,3,1,5,4};
int compar(void *arg1,void *arg2);

void main()  {
   qsort ( nums, 5, sizeof(int), compar);
}

int compar(void *arg1,void *arg2)  {
   if (* (int *) arg1 < (* (int *) arg2) return -1
   else if (* (int *) arg1 == (* (int *) arg2) return 0
   else return 1;
}
```

**Example Files:**
ex_qsort.c

**See Also:**
bsearch()

## rand( )

**Syntax:**
re=rand()

**Parameters:**
---

**Returns:**
A pseudo-random integer

**Function:**
The rand function returns a sequence of pseudo-random integers in the range of 0 to RAND_MAX.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**
```
int I;
I=rand();
```

**See Also:**
srand()

## rcv_buffer_bytes( )

**Syntax:**
value = rcv_buffer_bytes([**stream**]);

**Parameters:**
***stream*** – optional parameter specifying the stream defined in #USE RS232

**Returns:**
Number of bytes in receive buffer that still need to be retrieved

**Function:**
Function to determine the number of bytes in receive buffer that still need to be retrieve

**Availability:**
All Devices

**Requires:**
#USE RS232

**Examples:**
```
#USE_RS232(UART1,BAUD=9600,RECEIVE_BUFFER=100)
void main(void) {
   char c;
   if(rcv_buffer_bytes() > 10)
   c = getc();
}
```

**See Also:**
_USE_RS232( ), rcv_buffer_full( ), tx_buffer_full( ), tx_buffer_bytes( ), getc( ), putc( ) ,printf( ), setup_uart( ), putc_send( )

## rcv_buffer_full( )

**Syntax:**
value = rcv_buffer_full([stream]);

**Parameters:**
***stream*** – optional parameter specifying the stream defined in #USE RS232

**Returns:**
TRUE if receive buffer is full, FALSE otherwise

**Function:**
Function to test if the receive buffer is full

**Availability:**
All Devices

**Requires:**
#USE RS232

**Examples:**
```
#USE_RS232(UART1,BAUD=9600,RECEIVE_BUFFER=100)
void main(void) {
   char c;
   if(rcv_buffer_full())
   c = getc();
}
```

**See Also:**
_USE_RS232( ), rcv_buffer_full( ), tx_buffer_bytes( ), tx_buffer_bytes( ), getc( ), putc( ) ,printf( ), setup_uart( ), putc_send( )

## read_adc( ) [PCD] read_adc2( )

**Syntax:**
value = read_adc ([***mode***])
*[PCD]* value = read_adc2 ([***mode***])
*[PCD]* value=read_adc(**mode,[channel]**)

**Parameters:**
***mode* -** is an optional parameter. If used the values may be:
adc_start_and_read (continually takes readings, this is the default)
adc_start_only (starts the conversion and returns)
adc_read_only (reads last conversion result)

*[PCD]* **channel -** is an optional parameter for specifying the channel to start the conversion on and/or read the result from. If not specified will use channel specified in last call to **set_adc_channel()**, **read_adc()**, or **adc_done()**.

**Returns:**
Either a 8 or 16 bit int depending on #DEVICE ADC= directive.

**Function:**
This function will read the digital value from the analog to digital converter. Calls to setup_adc(), setup_adc_ports() and set_adc_channel() should be made sometime before this function is called. The range of the return value depends on number of bits in the chips A/D converter and the setting in the #DEVICE ADC= directive as follows:

| #DEVICE | 8 bit | 10 bit | 11 bit | 12 bit | 16 bit |
|---|---|---|---|---|---|
| **ADC=8** | 00-FF | 00-FF | 00-FF | 00-FF | 00-FF |
| **ADC=10** | x | 0-3FF | x | 0-3FF | x |
| **ADC=11** | x | x | 0-7FF | x | x |
| *[PCD]* **ADC=12** | | *[PCD]* 0-FFC | | *[PCD]* 0-FFF | |
| **ADC=16** | 0FF00 | 0-FFC0 | 0-FFEO | 0-FFF0 | 0-FFFF |

**Availability:**
This function is only available on devices with A/D hardware.
*[PCD]* Only available on devices with built in analog to digital converters.

**Requires:**
Pin constants are defined in the devices .h file

**Examples:**

```
setup_adc(  ADC_CLOCK_INTERNAL  );
setup_adc_ports( ALL_ANALOG );
set_adc_channel(1);
while ( input(PIN_B0) ) {
   delay_ms( 5000 );
   value = read_adc();
   printf("A/D value = %2x\n\r", value);
}
```

```
read_adc(ADC_START_ONLY);
sleep();
value=read_adc(ADC_READ_ONLY);
```

```
[PCD]
int16 value;
setup_adc_ports(sAN0|sAN1, VSS_VDD);
setup_adc(ADC_CLOCK_DIV_4|ADC_TAD_MUL_8);

while (TRUE)
{
   set_adc_channel(0);
   value = read_adc();
   printf("Pin AN0 A/C value = %LX\n\r", value);

   delay_ms(5000);

   set_adc_channel(1);
   read_adc(ADC_START_ONLY);
   ...
   value = read_adc(ADC_READ_ONLY);
   printf("Pin AN1 A/D value = %LX\n\r", value);
}
```

**Example Files:**
ex_admm.c, ex_14kad.c

**See Also:**
setup_adc(), set_adc_channel(), setup_adc_ports(), #DEVICE, ADC Overview

## read_bank( )

**Syntax:**
value = read_bank (***bank***, ***offset***)

**Parameters:**
***bank -*** is the physical RAM bank 1-3 (depending on the device)

***offset -*** is the offset into user RAM for that bank (starts at 0)

**Returns:**
8 bit int

**Function:**
Read a data byte from the user RAM area of the specified memory bank. This function may be used on some devices where full RAM access by auto variables is not efficient. For example, setting the pointer size to 5 bits on the PIC16C57 chip will generate the most efficient ROM code. However, auto variables can not be above 1Fh. Instead of going to 8 bit pointers, you can save ROM by using this function to read from 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:**
---

**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

**See Also:**
write_bank()

## read_calibration( )

**Syntax:**
value = read_calibration (***n***)

**Parameters:**
***n*** is an offset into calibration memory beginning at 0

**Returns:**
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:**
---

**Examples:**
```
fin = read_calibration(16);
```

**Example Files:**
ex_14kad.c with 14kcal.c

## read_calibration_memory( )

**Syntax:**
value = read_calibration_memory (cal_word)

**Parameters:**
**cal_word** - calibration word to read from calibration memory (1-16).

**Returns:**
unsigned int16 value read from calibration memory.

**Function:**
Allows for reading one of the calibration words from the calibration memory.

**Availability:**
This function is only available on MCP191xx devices.

**Requires:**
---

**Examples:**
```
CALWD1=read_calibration_memory(1);
```

**See Also:**
Program EEPROM Overview

## read_config_info( )

**Syntax:**
read_config_info(**[offset], ramPtr, count**)

**Parameters:**
**ramPTR** - is the destination pointer for the read results.

**count** - is the number of bytes to read.

**Offset** - is an optional parameter specifying the offset into the DCI memory to start reading from, offset default to zero if not used.

**Returns:**
---

**Function:**
Read **count** bytes from Device Configuration Area (DCI) memory and saves the values to **ramPtr**. The DCI region of memory contains read-only data about the device's configuration.

**Availability:**
Devices with a DCI memory region.

**Requires:**
---

**Examples:**

```
unsigned int16 EraseSize;
read_device_info(&EraseSize, 2);   //reads Erase Row Size from DCI
memory
```

**See Also:**
read_configuration_memory(), read_device_info(), Configuration Memory Overview

## read_configuration_memory( )

**Syntax:**
read_configuration_memory(**[offset], ramPtr, n**)

**Parameters:**
***ramPtr -*** is the destination pointer for the read results

***count* -** is an 8 bit integer

**offset -** is an optional parameter specifying the offset into configuration memory to start reading from, offset defaults to zero if not used.

**Returns:**
Undefined

**Function:**
Reads ***n*** bytes of configuration memory and saves the values to ***ramPtr***.
For **Enhanced16** devices function reads User ID, Device ID and configuration memory regions.

**Availability:**
All Devices

**Requires:**
---

**Examples:**

```
int data[6];
read_configuration_memory(data,6)
```

**See Also:**
write_configuration_memory(), read_program_memory(), Configuration Memory Overview, Configuration Memory Overview

## read_device_info( )

**Syntax:**
read_device_info(**[offset], ramPtr, count**)

**Parameters:**
**ramPTR** - is the destination pointer for the read results.

**count** - is the number of bytes to read.

**Offset** - is an optional parameter specifying the offset into the DIA memory to start reading from, offset default to zero if not used.

**Returns:**
---

**Function:**
Read **count** bytes from Device Information Area (DIA) memory and saves the values to **ramPtr**. The DIA region of memory contains read-only data used to identify the device.

**Availability:**
Devices with a DIA memory region.

**Requires:**
---

**Examples:**

```
unsigned int16 identifier[9];
read_device_info(identifier, 18);     //reads Unique Identifier from
DIA memory.
```

**See Also:**
read_configuration_memory(), read_config_info(), Configuration Memory Overview

## read_dmt( )

**Syntax:**
Value = read_dmt(**which**);

**Parameters:**
***which*** - an 8-bit constant indicating which DMT registers to read. The following defines are made in the device's header for selecting the register to read:
DMT_READ_COUNT // the current count
DMT_READ_MAX_VALUE // the value the count needs to reach for a DMT event to occur
DMT_READ_WINDOW_VALUE // the value the count needs to reach before it can be cleared

**Returns:**
An int32 value indicating the value that was read from the specified DMT registers.

**Function:**
Used to read the DMTCNT, DMTPSCNT and DMTPSINTV registers of the Deadman Timer (DMT) peripheral.

**Availability:**
---

**Requires:**
---

**Examples:**
```
  Value = read_dmt(DMT_READ_COUNT);
```

**See Also:**
clear_dmt(), disable_dmt(), enable_dmt(), dmt_status(), setup_dmt()

## read_eeprom( )

**Syntax:**
value = read_eeprom (***address)***

*[PCD]* value = read_eeprom (***address ,*** **[N])**
read_eeprom(***address***,***variable***)
read_eeprom(***address***, ***pointer***, ***N***)

**Parameters:**
***address -*** is an 8 bit or 16 bit int depending on the part

*[PCD]* ***N -*** specifies the number of EEPROM bytes to read

*[PCD]* ***variable -*** *a specified location to store EEPROM read results*

*[PCD]* ***pointer -*** *is a pointer to location to store EEPROM read results*

**Returns:**
An 8 bit int
*[PCD]* A 16 bit int

**Function:**
Reads a byte from the specified data EEPROM address. The address begins at 0 and the range depends on the part.
*[PCD]* By default the function reads a word from EEPROM at the specified address. The number of bytes to read can optionally be defined by argument N. If a variable is used as an argument, then EEPROM is read and the results are placed in the variable until the variable data size is full. Finally, if a pointer is used as an argument, then n bytes of EEPROM at the given address are read to the pointer.

**Availability:**
This command is only for parts with built-in EEPROMs

**Requires:**
---

**Examples:**

```
#define LAST_VOLUME  10
volume = read_EEPROM (LAST_VOLUME);
```

**See Also:**
write_eeprom(), erase_eeprom(), Data Eeprom Overview

## read_extended_ram( )

**Syntax:**
read_extended_ram(**page,address,data,count**);

**Parameters:**
**page** – the page in extended RAM to read from

**address** – the address on the selected page to start reading from

**data** – pointer to the variable to return the data to

**count** – the number of bytes to read (0-32768)

**Returns:**
Undefined

**Function:**
To read data from the extended RAM of the device.

**Availability:**
On devices with more then 30K of RAM

**Requires:**
---

**Examples:**

```
unsigned int8 data[8];
read_extended_ram(1,0x0000,data,8);
```

**See Also:**
Extended RAM Overview

## read_program_memory( )

**Syntax:**
READ_PROGRAM_MEMORY (***address***, ***dataptr***, ***count*** );

**Parameters:**
***address*** is 32 bits. The least significant bit should always be 0 in PCM.

***dataptr*** is a pointer to one or more bytes.

***count*** is a 8 bit integer on PIC16
***count*** is a 16 bit integer for PIC18 and dsPIC/PIC24

**Returns:**
Undefined

**Function:**
Reads ***count*** bytes from program memory at ***address*** to RAM at ***dataptr***.

**Availability:**
On devices with the ability to Read program memory.

**Requires:**
---

**Examples:**

```
char buffer[64];
read_program_memory(0x40000, buffer, 64);
```

**See Also:**
write program memory( ), External memory overview , Program Eeprom Overview

## read_high_speed_adc( )

**Syntax:**
read_high_speed_adc(**pair**,**mode**,**result)**; // Individual start and read or read only
read_high_speed_adc(**pair**,**result**); // Individual start and read
read_high_speed_adc(**pair**); // Individual start only
read_high_speed_adc(**mode**,**result**); // Global start and read or read only
read_high_speed_adc(**result**); // Global start and read
read_high_speed_adc(); // Global start only

**Parameters:**
**pair** – Optional parameter that determines which ADC pair number to start and/or read. Valid values are 0 to total number of ADC pairs. 0 starts and/or reads ADC pair AN0 and AN1, 1 starts and/or reads ADC pair AN2 and AN3, etc. If omitted then a global start and/or read will be performed.

**mode** – Optional parameter, if used the values may be:
**adc_start_and_read** (starts conversion and reads result)
**adc_start_only** (starts conversion and returns)
**adc_read_only** (reads conversion result)

**result** – Pointer to return ADC conversion too. Parameter is optional, if not used the **read_fast_adc()** function can only perform a start.

**Returns:**
Undefined

**Function:**
This function is used to start an analog to digital conversion and/or read the digital value when the conversion is complete. Calls to **setup_high_speed_adc()** and **setup_high_speed_adc_pairs()** should be made sometime before this function is called.

When using this function to perform an individual start and read or individual start only, the function assumes that the pair's trigger source was set to **individual_software_trigger**.

When using this function to perform a global start and read, global start only, or global read only. The function will perform the following steps:

1. Determine which ADC pairs are set for **global_software_trigger**
2. Clear the corresponding ready flags (if doing a start).
3. Set the global software trigger (if doing a start).
4. Read the corresponding ADC pairs in order from lowest to highest (if doing a read).
5. Clear the corresponding ready flags (if doing a read).

When using this function to perform a individual read only. The function can read the ADC result from any trigger source.

**Availability:**
Only on dsPIC33FJxxGSxxx devices

**Requires:**
Constants are define in the device .h file

**Examples:**

```
                                                    //Individual
start and read
int16 result[2];

setup_high_speed_adc(ADC_CLOCK_DIV_4);
setup_high_speed_adc_pair(0,  INDIVIDUAL_SOFTWARE_TRIGGER);
read_high_speed_adc(0, result);                 //starts
conversion for AN0
                                                //and AN1 and
stores result
                                                //in result[0]
and result[1]
                                                //Global start
and read
int16 result[4];

setup_high_speed_adc(ADC_CLOCK_DIV_4);
setup_high_speed_adc_pair(0, GLOBAL_SOFTWARE_TRIGGER);
setup_high_speed_adc_pair(4, GLOBAL_SOFTWARE_TRIGGER);
read_high_speed_adc(result);                    //starts
conversion for AN0, AN1,
                                                //AN8 and AN9 and
stores result in
                                                //result[0],
result //[1], result[2]
                                                //and result[3]
```

**See Also:**
setup_high_speed_adc(), setup_high_speed_adc_pair(), high_speed_adc_done()

## read_program_memory( )

**Syntax:**
value = read_program_eeprom (***address***)

**Parameters:**
***address -*** is 16 bits on PCM parts and 32 bits on PCH parts

**Returns:**
16 bits

**Function:**
Reads data from the program memory

**Availability:**
Only devices that allow reads from program memory

**Requires:**
---

**Examples:**

```
checksum = 0;
for(i=0;i<8196;i++)
   checksum^=read_program_eeprom(i);
printf("Checksum is %2X\r\n",checksum);
```

**See Also:**
write_program_eeprom(), write_eeprom(), read_eeprom(), Program Eeprom Overview

## read_program_memory( )

**Syntax:**
READ_PROGRAM_MEMORY (***address***, ***dataptr***, ***count*** );

**Parameters:**
***address*** is 32 bits. The least significant bit should always be 0 in PCM.

***dataptr*** is a pointer to one or more bytes.

***count*** is a 8 bit integer on PIC16
***count*** is a 16 bit integer for PIC18 and dsPIC/PIC24

**Returns:**
Undefined

**Function:**
Reads ***count*** bytes from program memory at ***address*** to RAM at ***dataptr***.

**Availability:**
On devices with the ability to Read program memory.

**Requires:**
---

**Examples:**

```
char buffer[64];
read_program_memory(0x40000, buffer, 64);
```

**See Also:**
write program memory( ), External memory overview , Program Eeprom Overview

## read_program_memory8( )

**Syntax:**
READ_PROGRAM_MEMORY8 (***address***, ***dataptr***, ***count*** );

**Parameters:**
***address*** is 16 bits to start reading data from the program memory.

***dataptr*** is a pointer to an array of bytes to store read data to.

***count*** is the number of bytes to read from program memory.

**Returns:**
Undefined

**Function:**
Reads ***count*** bytes from program memory. This function only reads the least significant byte from each address in program memory. See **read_program_memory()** for a function that can read all the data from each address in program memory.

**Availability:**
Only on PCM devices with the ability to Read program memory.

**Requires:**
---

**Examples:**

```
read_program_memory8(Address, Data, 128);
```

**See Also:**
read_program_memory( ), write program memory( ), write_program_memory8( ), Program Eeprom Overview

## read_rom_memory( )

**Syntax:**
read_rom_memory (***address***, ***dataptr***, ***count*** );

**Parameters:**
***address -*** is 32 bits. The least significant bit should always be 0.
***dataptr -*** is a pointer to one or more bytes.
***count -*** is a 16 bit integer

**Returns:**
Undefined

**Function:**
Reads ***count*** bytes from program memory at ***address*** to ***dataptr***.
*[PCD]* 24 bit program instruction size, 3 bytes are read from each address location

**Availability:**
Only devices that allow reads from program memory

**Requires:**
---

**Examples:**

```
char buffer[64];
read_program_memory(0x40000, buffer, 64);
```

**See Also:**
write_program_eeprom() , write_eeprom(), read_eeprom(), Program eeprom overview

## read_sd_adc( )

**Syntax:**
value = read_sd_adc();

**Parameters:**
---

**Returns:**
A signed 32 bit int

**Function:**
To poll the SDRDY bit and if set return the signed 32 bit value stored in the SD1RESH and SD1RESL registers, and clear the SDRDY bit. The result returned depends on settings made with the **setup_sd_adc()** function, but will always be a signed int32 value with the most significant bits being meaningful. Refer to Section 66, 16-bit Sigma-Delta

A/D Converter, of the PIC24F Family Reference Manual for more information on the module and the result format.

**Availability:**
Only devices with a Sigma-Delta Analog to Digital Converter (SD ADC) module

**Requires:**
---

**Examples:**

```
value = read_sd_adc()
```

**See Also:**
setup_sd_adc(), set_sd_adc_calibration(), set_sd_adc_channel()

## realloc( )

**Syntax:**
realloc (***ptr***, ***size***)

**Parameters:**
***ptr -*** is a null pointer or a pointer previously returned by calloc or malloc or realloc function, size is an integer representing the number of byes to be allocated.

**Returns:**
A pointer to the possibly moved allocated memory, if any. Returns null otherwise.

**Function:**
The **realloc** function changes the size of the object pointed to by the **ptr** to the size specified by the size. The contents of the object shall be unchanged up to the lesser of new and old sizes. If the new size is larger, the value of the newly allocated space is indeterminate. If **ptr** is a null pointer, the **realloc** function behaves like **malloc** function for the specified size. If the **ptr** does not match a pointer earlier returned by the **calloc**, **malloc** or **realloc**, or if the space has been deallocated by a call to free or **realloc** function, the behavior is undefined. If the space cannot be allocated, the object pointed to by **ptr** is unchanged. If size is zero and the **ptr** is not a null pointer, the object is to be freed.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlibm.h>

**Examples:**

```
int * iptr;
iptr=malloc(10);
realloc(iptr,20)              // iptr will point to a block of
memory of
                              // 20 bytes, if available
```

**See Also:**
malloc(), free(), calloc()

## release_io( )

**Syntax:**
release_io();

**Parameters:**
---

**Returns:**
---

**Function:**
The function is used to release the I/O on devices that have woken up from deep sleep.

**Availability:**
Devices with a Deep Sleep Watch Dog Timer (DSWDT) peripheral.

**Requires:**
---

**Examples:**

```
restart=restart_cause();

switch(restart)
{
  case RTC_FROM_DS:
  case DSWDT_FROM_DS:
  case ULPWU_FROM_DS:
  case EXT_FROM_DS:
      release_io();
      break;
```

```
}
```

**See Also:**
sleep()

## reset_cpu( )

**Syntax:**
reset_cpu()

**Parameters:**
---

**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 PIC18.

**Availability:**
All Devices

**Requires:**
---

**Examples:**

```
if(checksum!=0)
   reset_cpu();
```

## restart_cause( )

**Syntax:**
value = restart_cause()

**Parameters:**
---

**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

*[PCD]* reset_power_up, restart_brownout, restart_wdt and sestart_mclr, wdt_from_sleep

**Function:**
Returns the cause of the last processor reset.

*[PCD]* In order for the result to be accurate, it should be called immediately in main().

**Availability:**
All Devices

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
switch ( restart_cause() ) {
     case WDT_FROM_SLEEP:
   case WDT_TIMEOUT:
     handle_error();
}
```
*[PCD]*
```
switch ( restart_cause() ) {
     case RESTART_BROWNOUT:
   case RESTART_WDT:
   case RESTART_MCLR:
     handle_error();
}
```

**Example Files:**
ex_wdt.c

**See Also:**
restart_wdt(), reset_cpu()

## restart_wdt( )

**Syntax:**
restart_wdt()

**Parameters:**
---

**Returns:**
---

**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

**See Also:**
#FUSES, setup_wdt(), WDT or Watch Dog Timer Overview

## rotate_left( )

**Syntax:**
rotate_left (***address***, ***bytes***)

**Parameters:**
***address -*** is a pointer to memory
***bytes -*** is a count of the number of bytes to work with

**Returns:**
Undefined

**Function:**
Rotates a bit through an array or structure. The address may be an array identifier or an address to a byte or structure (such as &data). Bit 0 of the lowest BYTE in RAM is considered the LSB.

**Availability:**
All Devices

**Requires:**
---

**Examples:**

```
x = 0x86;
rotate_left( &x, 1);          // x is now 0x0d
```

**See Also:**
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:**
---

**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
```

**See Also:**
rotate_left(), shift_left(), shift_right()

## rtc_alarm_read( )

**Syntax:**
rtc_alarm_read(&***datetime***);

**Parameters:**
***datetime***- A structure that will contain the values to be written to the alarm in the RTCC module.

Structure used in read and write functions are defined in the device header file as **rtc_time_t**

**Returns:**
Void

**Function:**
Reads the date and time from the alarm in the RTCC module to structure ***datetime***.

**Availability:**
Devices that an RTCC module

**Requires:**
---

**Examples:**

```
rtc_alarm_read(&datetime);
```

**See Also:**
rtc_read(), rtc_alarm_read(), rtc_alarm_write(), setup_rtc_alarm(), rtc_write(), setup_rtc()

## rtc_alarm_write( )

**Syntax:**
rtc_alarm_write(&***datetime***);

**Parameters:**
***datetime***- A structure that will contain the values to be written to the alarm in the RTCC module.

Structure used in read and write functions are defined in the device header file as **rtc_time_t**

**Returns:**
Void

**Function:**
Write the date and time from the alarm in the RTCC module to structure ***datetime***.

**Availability:**
Devices that an RTCC module

**Requires:**
---

**Examples:**

```
rtc_alarm_write(&datetime);
```

**See Also:**
rtc_read(), rtc_alarm_read(), rtc_alarm_write(), setup_rtc_alarm(), rtc_write(), setup_rtc()

## rtc_read( )

**Syntax:**
rtc_read(&***datetime***);

**Parameters:**
***datetime***- A structure that will contain the values returned by the RTCC module.

Structure used in read and write functions are defined in the device header file as **rtc_time_t**

**Returns:**
Void

**Function:**
Reads the current value of Time and Date from the RTCC module and stores the structure date time.

**Availability:**
Devices that have a Real-Time Clock and Calendar (RTCC) module.

**Requires:**
---

**Examples:**

```
rtc_read(&datetime);
```

**Example Files:**
ex_rtcc.c

**See Also:**
rtc_read(), rtc_alarm_read(), rtc_alarm_write(), setup_rtc_alarm(), rtc_write(), setup_rtc()

## [PCD] rtc_status( )

**Syntax:**
Status = rtc_status();

**Parameters:**
---

**Returns:**
An int8 value indicating the status of the RTCC module. See the device's header file for constants that can be and'ed with return value to determine that state of the individual status bits.

**Function:**
Used to determine the status of the RTCC module.

**Availability:**
Devices that have a Real-Time Clock and Calendar (RTCC) with Timestamp module.

**Requires:**
---

**Examples:**

```
rtc_time_t TimeStamp;
rtc_tsa_read(&TimeStamp);       //read Timestamp A registers
rtc_tsb_read(&TimeStamp);       //read Timestamp B registers
```

**See Also:**
setup_rtc(), setup_rtc_alarm(), rtc_read(), rtc_write(), rtc_alarm_read(), rtc_alarm_write( ), rtc_tsx_read()

## [PCD] rtc_tsx_read()

**Syntax:**
rtc_tsa_read(&**timestamp**);
rtc_tsb_read(&**timestamp**);

**Parameters:**
**timestamp** - a structure of **rtc_time_t** to return the timestamp value.

**Returns:**
---

**Function:**
Used to read the Timestamp A and Timestamp B registers and converts them to be compatible with the **rtc_time_t** structure.

**Availability:**
Devices that have a Real-Time Clock and Calendar (RTCC) with Timestamp module.

**Requires:**
---

**Examples:**

```
rtc_time_t TimeStamp;
rtc_tsa_read(&TimeStamp);       //read Timestamp A registers
rtc_tsb_read(&TimeStamp);       //read Timestamp B registers
```

**See Also:**
setup_rtc(), setup_rtc_alarm(), rtc_read(), rtc_write(), rtc_alarm_read(), rtc_alarm_write( ), rtc_status()

## rtc_write( )

**Syntax:**
rtc_write(&***datetime***);

**Parameters:**
***datetime***- A structure that will contain the values to be written to the RTCC module.

Structure used in read and write functions are defined in the device header file as **rtc_time_t**

**Returns:**
Void

**Function:**
Writes the date and time to the RTCC module as specified in the structure date time.

**Availability:**
Devices that an RTCC module

**Requires:**
---

**Examples:**

```
rtc_write(&datetime);
```

**Example Files:**
ex_rtcc.c

**See Also:**
rtc_read() , rtc_alarm_read() , rtc_alarm_write() , setup_rtc_alarm() , rtc_write(), setup_rtc()

## rtos_await( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_await (***expre***)

**Parameters:**
***expre*** is a logical expression

**Returns:**
---

**Function:**
This function can only be used in an RTOS task. This function waits for ***expre*** to be true before continuing execution of the rest of the code of the RTOS task. This function allows other tasks to execute while the task waits for ***expre*** to be true.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
rtos_await(kbhit());
```

**See Also:**

## rtos_disable( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_disable ***(task)***

**Parameters:**
***task -*** *is the identifier of a function that is being used as an RTOS task*

**Returns:**
---

**Function:**
This function disables a task which causes the task to not execute until enabled by rtos_enable(). All tasks are enabled by default.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
rtos_disable(toggle_green);
```

**See Also:**
rtos enable()

## rtos_enable( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_enable ***(task)***

**Parameters:**
***task -*** *is the identifier of a function that is being used as an RTOS task*

**Returns:**
---

**Function:**
This function enables a task to execute at it's specified rate.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
rtos_enable(toggle_green);
```

**See Also:**
rtos disable()

## rtos_msg_poll( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
i = rtos_msg_poll()

**Parameters:**
---

**Returns:**
An integer that specifies how many messages are in the queue

**Function:**
This function can only be used inside an RTOS task. This function returns the number of messages that are in the queue for the task that the rtos_msg_poll() function is used in.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
if(rtos_msg_poll())
```

**See Also:**
rtos msg send(), rtos msg read()

## rtos_msg_read( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
b = rtos_msg_read()

**Parameters:**
---

**Returns:**
A byte that is a message for the task

**Function:**
This function can only be used inside an RTOS task. This function reads in the next (message) of the queue for the task that the rtos_msg_read() function is used in.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
if(rtos_msg_poll()) {
   b = rtos_msg_read();
```

**See Also:**
rtos msg poll(), rtos msg send()

## rtos_msg_send( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_msg_send(***task, byte***)

**Parameters:**
***task -*** is the identifier of a function that is being used as an RTOS task

***byte -*** is the byte to send to ***task*** as a message

**Returns:**
---

**Function:**
This function can be used anytime after rtos_run() has been called.
This function sends a byte long message (***byte***) to the task identified by ***task***.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
if(kbhit())
{
   rtos_msg_send(echo, getc());
}
```

**See Also:**
rtos_msg_poll(), rtos_msg_read()

## rtos_overrun( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_overrun(***[task]***)

**Parameters:**
***task -*** is an optional parameter that is the identifier of a function that is being used as an RTOS task

**Returns:**
A 0 (FALSE) or 1 (TRUE)

**Function:**
This function returns TRUE if the specified task took more time to execute than it was allocated. If no task was specified, then it returns TRUE if any task ran over it's alloted execution time.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
rtos_overrun();
```

## rtos_run( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_run( )

**Parameters:**
---

**Returns:**
---

**Function:**
This function begins the execution of all enabled RTOS tasks. This function controls the execution of the RTOS tasks at the allocated rate for each task. This function will return only when rtos_terminate() is called.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
rtos_run();
```

**See Also:**
rtos terminate()

## rtos_signal( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_signal ***(sem)***

**Parameters:**
***sem*** is a global variable that represents the current availability of a shared system resource (a semaphore)

**Returns:**
---

**Function:**
This function can only be used by an RTOS task. This function increments ***sem*** to let waiting tasks know that a shared resource is available for use.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
rtos_signal(uart_use);
```

**See Also:**
rtos wait()

## rtos_stats( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_stats(***task,&stat***)

**Parameters:**
***task -*** is the identifier of a function that is being used as an RTOS task.
***stat -*** is a structure containing the following:

```
      struct rtos_stas_struct {
         unsigned int32 task_total_ticks;   //number of ticks the
task has used
         unsigned int16 task_min_ticks;     //the minimum number
of ticks used
         unsigned int16 task_max_ticks;     //the maximum number
of ticks used
         unsigned int16 hns_per_tick;       //us =
(ticks*hns_per_tick)/10
```

**Returns:**
Undefined

**Function:**
This function returns the statistic data for a specified ***task***.

**Availability:**
All Devices

**Requires:**
#USE RTOS(statistics)

**Examples:**

```
  rtos_stats(echo, &stats);
```

**See Also:**

## rtos_terminate( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_terminate()

**Parameters:**
---

**Returns:**
---

**Function:**
This function ends the execution of all RTOS tasks. The execution of the program will continue with the first line of code after the rtos_run() call in the program. (This function causes rtos_run() to return.)

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
rtos_terminate()
```

**See Also:**
rtos run()

## rtos_wait( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_wait (***sem****)*

**Parameters:**
***sem*** is a global variable that represents the current availability of a shared system resource (a semaphore)

**Returns:**
---

**Function:**
This function can only be used by an RTOS task. This function waits for ***sem*** to be greater than 0 (shared resource is available), then decrements ***sem*** to claim usage of the shared resource and continues the execution of the rest of the code  the RTOS task. This function allows other tasks to execute while the task waits for the shared resource to be available.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
rtos_wait(uart_use)
```

**See Also:**
rtos signal()

## rtos_yield( )

**The RTOS is only included in the PCW, PCWH and PCWHD software packages.**

**Syntax:**
rtos_yield()

**Parameters:**
---

**Returns:**
-----

**Function:**
This function can only be used in an RTOS task. This function stops the execution of the current task and returns control of the processor to **rtos_run()**. When the next task executes, it will start its execution on the line of code after the **rtos_yield()**.

**Availability:**
All Devices

**Requires:**
#USE RTOS

**Examples:**

```
void yield(void)
{
     printf("Yielding...\r\n");
     rtos_yield();
```

```
      printf("Executing code after yield\r\n");
}
```

## set_adc_channel( ) set_adc2_channel( )

**Syntax:**
set_adc_channel (***chan*** [,***neg***]))
*[PCD]* set_adc_channel(chan, [differential]) //dsPIC33EPxxGSxxx only
*[PCD]* set_adc2_channel(***chan***)

**Parameters:**
***chan*** is the channel number to select. Channel numbers start at 0 and are labeled in the data sheet AN0, AN1. For devices with a differential ADC it sets the positive channel to use.

***neg*** is optional and is used for devices with a differential ADC only. It sets the negative channel to use, channel numbers can be 0 to 6 or VSS. If no parameter is used the negative channel will be set to VSS by default.

**Returns:**
Undefined
*[PCD]* ***differential*** is an optional parameter to specify if channel is differential or single-ended. TRUE is differential and FALSE is single-ended. Only available for dsPIC3EPxxGSxxx family.

**Function:**
Specifies the channel to use for the next read_adc() call. Be aware that you must wait a short time after changing the channel before you can get a valid read. The time varies depending on the impedance of the input source. In general 10us is good for most applications. You need not change the channel before every read if the channel does not change.

**Availability:**
This function is only available on devices with A/D hardware.
*[PCD]* Only available on devices with built in analog to digital converters

**Requires:**
---

**Examples:**

```
set_adc_channel(2);
delay_us(10);
value = read_adc();
```

**Example Files:**
ex_admm.c

**See Also:**
read_adc(), setup_adc(), setup_adc_ports(), ADC Overview

## set_adc_trigger( )

**Syntax:**
set_adc_trigger (**trigger**)

**Parameters:**
***trigger*** - ADC trigger source. Constants defined in device's header, see the device's .h file for all options. Some typical options include:

ADC_TRIGGER_DISABLED
ADC_TRIGGER_ADACT_PIN
ADC_TRIGGER_TIMER1
ADC_TRIGGER_CCP1

**Returns:**
Undefined

**Function:**
Sets the Auto-Conversion trigger source for the Analog-to-Digital Converter with Computation (ADC2) Module.

**Availability:**
All devices with an ADC2 Module

**Requires:**
Constants defined in the device's .h file

**Examples:**

```
set_adc_trigger(ADC_TRIGGER_TIMER1);
```

**See Also:**
ADC Overview, setup_adc(), setup_adc_ports(), set_adc_channel(), read_adc(), #DEVICE, adc_read(), adc_write(), adc_status()

## set_analog_pins( )

**Syntax:**
set_analog_pins(**pin, pin, pin, ...**)

**Parameters:**
***pin*** - pin to set as an analog pin. Pins are defined in the device's .h file. The actual value is a bit address. For example, bit 3 of port A at address 5, would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43

**Returns:**
Undefined

**Function:**
To set which pins are analog and digital. Usage of function depends on method device has for setting pins to analog or digital. For devices with ANSELx, x being the port letter, registers the function is used as described above. For all other devices the function works the same as **setup_adc_ports()** function.

**Availability:**
On all devices with an Analog to Digital Converter

**Requires:**
---

**Examples:**

```
set_analog_pins(PIN_A0,PIN_A1,PIN_E1,PIN_B0,PIN_B5);
```

**See Also:**
setup_adc_reference(), set_adc_channel(), read_adc(), setup_adc(), setup_adc_ports(), ADC Overview

## scanf( ) fscanf( )

**Syntax:**
scanf(**cstring**);
scanf(**cstring, values...**)
fscanf(**stream, cstring, values...**

**Parameters:**
***cstring*** is a constant string.

**values** is a list of variables separated by commas.

**stream** is a stream identifier

**Returns:**
0 if a failure occurred, otherwise it returns the number of conversion specifiers that were read in, plus the number of constant strings read in.

**Function:**
Reads in a string of characters from the standard RS-232 pins and formats the string according to the format specifiers. The format specifier character (%) used within the string indicates that a conversion specification is to be done and the value is to be saved into the corresponding argument variable. A %% will input a single %. Formatting rules for the format specifier as follows:

If fscanf() is used, then the specified stream is used, where scanf() defaults to STDIN (the last USE RS232).

Format:
The format takes the generic form %nt. **n** is an option and may be 1-99 specifying the field width, the number of characters to be inputted. **t** is the type and maybe one of the following:

**c** Matches a sequence of characters of the number specified by the field width (1 if no field width is specified). The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence.

**s** Matches a sequence of non-white space characters. The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence and a terminating null character, which will be added automatically.

**u** Matches an unsigned decimal integer. The corresponding argument shall be a pointer to an unsigned integer.

**Lu** Matches a long unsigned decimal integer. The corresponding argument shall be a pointer to a long unsigned integer.

**d** Matches a signed decimal integer. The corresponding argument shall be a pointer to a signed integer.

**Ld** Matches a long signed decimal integer. The corresponding argument shall be a pointer to a long signed integer.

**o** Matches a signed or unsigned octal integer. The corresponding argument shall be a pointer to a signed or unsigned integer.

**Lo** Matches a long signed or unsigned octal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.

**x or X** Matches a hexadecimal integer. The corresponding argument shall be a pointer to a signed or unsigned integer.

**Lx or LX** Matches a long hexadecimal integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.

**i** Matches a signed or unsigned integer. The corresponding argument shall be a pointer to a signed or unsigned integer.

**Li** Matches a long signed or unsigned integer. The corresponding argument shall be a pointer to a long signed or unsigned integer.

**f,g or e** Matches a floating point number in decimal or exponential format. The corresponding argument shall be a pointer to a float.

**[** Matches a non-empty sequence of characters from a set of expected characters. The sequence of characters included in the set are made up of all character following the left bracket ([) up to the matching right bracket (]). Unless the first character after the left bracket is a **^**, in which case the set of characters contain all characters that do not appear between the brackets. If a **-** character is in the set and is not the first or second, where the first is a **^**, nor the last character, then the set includes all characters from the character before the **-** to the character after the **-**.

For example, %[a-z] would include all characters from **a** to **z** in the set and %[^a-z] would exclude all characters from **a** to **z** from the set. The corresponding argument shall be a pointer to the initial character of an array long enough to accept the sequence and a terminating null character, which will be added automatically.

**n** Assigns the number of characters read thus far by the call to scanf() to the corresponding argument. The corresponding argument shall be a pointer to an unsigned integer.

An optional assignment-suppressing character (*****) can be used after the format specifier to indicate that the conversion specification is to be done, but not saved into a corresponding variable. In this case, no corresponding argument variable should be passed to the scanf() function.

A string composed of ordinary non-white space characters is executed by reading the next character of the string. If one of the inputted characters differs from the string, the function fails and exits. If a white-space character precedes the ordinary non-white space characters, then white-space characters are first read in until a non-white space character is read.

White-space characters are skipped, except for the conversion specifiers **[**, **c** or **n**, unless a white-space character precedes the **[** or **c** specifiers.

**Availability:**
All Devices

**Requires:**
#USE RS232

**Examples:**
```
char name[2-];
unsigned int8 number;
signed int32 time;

if(scanf("%u%s%ld",&number,name,&time))
   printf"\r\nName: %s, Number: %u, Time: %ld",name,number,time
```

**See Also:**
RS232 I/O Overview, getc(), putc(), printf()

## [PCD] sent_getd( )

**Syntax:**
data = sent_getd(**module**);

**Parameters:**
***module*** - the SENT peripheral to setup, 1 or 2 for most devices.

**Returns:**
The data read by the SENT peripheral when it is setup as a receiver. The data type SENT_DATA_TYPE is defined in the device's header file for organizing the nibble data.

**Function:**
Gets data from the Single-Edge Nibble Transmission (SENT) peripheral's data registers.

**Availability:**
Devices with a SENT peripheral.

**Requires:**
---

**Examples:**

```
SENT_DATA_TYPE Data;
Data = sent_getd(1);
```

**Example Files:**
ex_sent_transmitter.c, ex_sent_receiver.c

**See Also:**
sent_putd( ), setup_sent( ),sent_status( )

## [PCD] sent_putd( )

**Syntax:**
sent_putd(**module, data**);

**Parameters:**
***module*** - the SENT peripheral to setup, 1 or 2 for most devices.

***data*** - the data to transmit when SENT peripheral is setup as a transmitter. The data type SENT_DATA_TYPE is defined in the device's header file for organizing the nibble data.

**Returns:**
---

**Function:**
Puts data for transmission into the Single-Edge Nibble Transmission (SENT) peripheral's data registers.

**Availability:**
Devices with a SENT peripheral.

**Requires:**
---

**Examples:**

```
SENT_DATA_TYPE Data;
sent_putd(1, Data);
```

**Example Files:**
ex_sent_transmitter.c, ex_sent_receiver.c

**See Also:**
sent_getd( ),setup_sent( ), sent_status( )

## [PCD] sent_status( )

**Syntax:**
status = sent_status(**module**);

**Parameters:**
***module*** - the SENT peripheral to setup, 1 or 2 for most devices.

**Returns:**
The status of the SENT peripheral. See device's header file for constants that can be and'ed with return value to determine which status flags are set.

**Function:**
Gets status from the Single-Edge Nibble Transmission (SENT) peripheral's status register.

**Availability:**
Devices with a SENT peripheral.

**Requires:**
---

**Examples:**

```
unsigned int8 status;
status = sent_status(1);
```

**Example Files:**
ex_sent_transmitter.c, ex_sent_receiver.c

**See Also:**
sent_putd( ), sent_getd( ), setup_sent( )

## set_ccp1_compare_time( ) set_ccp2_compare_time( ) set_ccp3_compare_time( ) set_ccp5_compare_time( ) set_ccp5_compare_time( )

**Syntax:**
set_ccpx_compare_time(**time**);
set_ccpx_compare_time(**timeA, timeB**)

**Parameters:**
**time -** may be a 16 or 32-bit constant or varaible. If 16-bit, it sets the CCPxRAL register to the value time and CCPxRBL to zero; used for single edge output compare mode set for 16-bit timer mode. If 32-bit, it sets the CCPxRAL and CCPxRBL register to the value time, CCPxRAL least significant word and CCPRBL most significant word; used for single edge output compare mode set for 32-bit timer mode.

**timeA** - is a 16-bit constant or variable to set the CCPxRAL register to the value of timeA, used for dual edge output c ompare and PWM modes.

**timeB** - is a 16-bit constant or variable to set the CCPxRBL register to the value of timeB, used for dual edge output compare and PWM modes.

**Returns:**
Undefined

**Function:**
This function sets the compare value for the CCP module. If the CCP module is performing a single edge compare in 16-bit mode, then the CCPxRBL register is not used. If 32-bit mode, the CCPxRBL is the most significant word of the compare time. If the CCP module is performing dual edge compare to generate an output pulse, then timeA, CCPxRAL register, signifies the start of the pulse, and timeB, CCPxRBL register signifies the pulse termination time.

**Availability:**
Available only on PIC24FxxKMxxx family of devices with a MCCP and/or SCCP modules

**Requires:**
---

**Examples:**

```
setup_ccp1(CCP_COMPARE_PULSE);
set_timer_period_ccp1(800);
set_ccp1_compare_time(200,300);     //generate a pulse starting
at time
```

```
                                        // 200 and ending at time
300
```

**See Also:**
set_pwmX_duty(), setup_ccpX(), set_timer_period_ccpX(), set_timer_ccpX(), get_timer_ccpX(), get_capture_ccpX(), get_captures32_ccpX()

## set_cog_blanking( ) set_cog2_blanking( ) set_cog3_blanking( ) set_cog4_blanking( )

**Syntax:**
set_cog_blanking(falling_time, rising_time);

**Parameters:**
**falling time** - sets the falling edge blanking time.

**rising time** - sets the rising edge blanking time

**Returns:**
---

**Function:**
To set the falling and rising edge blanking times on the Complementary Output Generator (COG) module.
The time is based off the source clock of the COG module, the times are either a 4-bit or 6-bit value, depending on the device, refer to the
device's datasheet for the correct width.

**Availability:**
All devices with a COG module

**Requires:**
---

**Examples:**

```
set_cog_blanking(10,10);
```

**See Also:**
setup_cog(), set_cog_phase(), set_cog_dead_band(), cog_status(), cog_restart()

## set_cog_dead_band( ) set_cog2_dead_band( ) set_cog3_dead_band( ) set_cog4_dead_band( )

**Syntax:**
set_cog_dead_band(falling_time, rising_time);

**Parameters:**
**falling time** - sets the falling edge dead-band time.

**rising time** - sets the rising edge dead-band time.

**Returns:**
---

**Function:**
To set the falling and rising edge dead-band times on the Complementary Output Generator (COG) module.
The time is based off the source clock of the COG module, the times are either a 4-bit or 6-bit value, depending on the device, refer to the
device's datasheet for the correct width.

**Availability:**
All devices with a COG module

**Requires:**
---

**Examples:**

```
set_cog_dead_band(16,32);
```

**See Also:**
setup_cog(), set_cog_phase(), set_cog_blanking(), cog_status(), cog_restart()

## set_cog_phase( ) set_cog2_phase( ) set_cog3_phase( ) set_cog4_phase( )

**Syntax:**
set_cog_phase(rising_time);
set_cog_phase(falling_time, rising_time);

**Parameters:**
**falling time** - sets the falling edge phase time.

**rising time** - sets the rising edge phase time.

**Returns:**
---

**Function:**
To set the falling and rising edge phase times on the Complementary Output Generator (COG) module.
The time is based off the source clock of the COG module, the times are either a 4-bit or 6-bit value, depending on the device.
Some devices only have a rising edge delay, refer to the device's datasheet.

**Availability:**
All devices with a COG module

**Requires:**
---

**Examples:**

```
set_cog_phase(10,10);
```

**See Also:**
setup_cog(), set_cog_dead_band(), set_cog_blanking(), cog_status(), cog_restart()

## set_compare_time( )

**Syntax:**
set_compare_time(***x***, ***time***])
*[PCD]* set_compare_time(***x***, o***cr, [ocrs]***])

**Parameters:**
***x -*** is 1-8 and defines which output compare module to set time for.

***time -*** is the compare time for the primary compare register.
*[PCD]*
***x -*** is 1-16 and defines which output compare module to set time for.

***ocr -*** is the compare time for the primary compare register.

***ocrs -*** is the optional compare time for the secondary register. Used for dual compare mode.

**Returns:**
---

**Function:**
This function sets the compare value for the CCP module.
*[PCD]* This function sets the compare value for the output compare module. If the output compare module is to perform only a single compare than the ***ocrs*** register is not used. If the output compare module is using double compare to generate an output pulse, the ***ocr*** signifies the start of the pulse and ***ocrs*** defines the pulse termination time.

**Availability:**
All devices with a CCP module
*[PCD]* All devices with Output Compare modules

**Requires:**
---

**Example Files:**
ex_ccp1s.c

*[PCD]* **Example File:**

```
                                                    // Pin OC1 will be set
  when
                                                     //timer 2 is equal to
  0xF000
  setup_timer2(TMR_INTERNAL | TIMER_DIV_BY_8);
  setup_compare_time(1, 0xF000);
  setup_compare(1, COMPARE_SET_ON_MATCH | COMPARE_TIMER2);
```

**See Also:**
get_capture( ), setup_ccpx( )
*[PCD]* Output Compare

## set_dedicated_adc_channel( )

**Syntax:**
set_dedicated_adc_channel(**core,channel, [differential]**);

**Parameters:**
**core** - the dedicated ADC core to setup

**channel** - the channel assigned to the specified ADC core. Channels are defined in the device's .h file as follows:
ADC_CHANNEL_AN0
ADC_CHANNEL_AN7

ADC_CHANNEL_PGA1
ADC_CHANNEL_AN0ALT
ADC_CHANNEL_AN1
ADC_CHANNEL_AN18
ADC_CHANNEL_PGA2
ADC_CHANNEL_AN1ALT
ADC_CHANNEL_AN2
ADC_CHANNEL_AN11
ADC_CHANNEL_VREF_BAND_GAP
ADC_CHANNEL_AN3
ADC_CHANNEL_AN15

Not all of the above defines can be used with all the dedicated ADC cores. Refer to the device's header for which can be used with each dedicated ADC core.

**differential** - optional parameter to specify if channel is differential or single-ended. TRUE is differential and FALSE is single-ended.

**Returns:**
Undefined

**Function:**
Sets the channel that will be assigned to the specified dedicated ADC core. Function does not set the channel that will be read with the next call to read_adc(), use set_adc_channel() or read_adc() functions to set the channel that will be read.

**Availability:**
Only dsPIC33EPxxGSxxx family of devices

**Requires:**
---

**Examples:**

```
setup_dedicated_adc_channel(0,ADC_CHANNEL_AN0);
```

**See Also:**
setup_adc(), setup_adc_ports(), set_adc_channel(), read_adc(), adc_done(), setup_dedicated_adc(), ADC Overview

## set_hspwm_event( )  set_hspwm_secondary_event( )

**Syntax:**
set_hs hspwm_event(**settings, compare_time**);
set_hswpm_secondary_event(**settings, compare_time**);  //if available

**Parameter:**
**settings** - special event timer setting or'd with a value from 1 to 16 to set the prescaler.
The following are the settings available for the special event time:
· HSPWM_SPECIAL_EVENT_INT_ENABLED
· HSPWM_SPECIAL_EVENT_INT_DISABLED

**compare_time** - the compare time for the special event to occur

**Returns:**
---

**Function:**
Sets the specified High Speed PWM unit.

**Availability:**
Only on devices with a built-in High Speed PWM module
(dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx,
and dsPIC33EVxxxGMxxx devices)

**Requires:**
Constants are defined in the device's .h file

**Examples:**
---

**See Also:**
setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_duty(),
setup_hspwm_blanking(), setup_hspwm_trigger(), set_hspwm_override(),
get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock()
setup_hspwm(), setup_hspwm_secondary()

## set_hspwm_duty( )

**Syntax:**
set_hspwm_duty**(duty**);
set_hspwm_duty(**unit,primary, [secondary]**;

**Parameters:**
**duty** - A 16-bit constant or variable to set the master duty cycle

**unit** - The High Speed PWM unit to set.

**primary** - A 16-bit constant or variable to set the primary duty cycle.

**secondary** - An optional 16-bit constant or variable to set the secondary duty cycle. Secondary duty cycle is only used in Independent PWM mode. Not available on all devices, refer to the device datasheet for availability.

**Returns:**
Undefined

**Function:**
Sets the specified High Speed PWM unit.

**Availability:**
Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**
---

**Examples:**

```
set_hspwm_duty(0x7FFF);      //sets the high speed PWM
                            //master duty cycle
set_hspwm_duty(1, 0x3FFF);  //sets unit 1's primary duty cycle
```

**See Also:**
setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_event(), setup_hspwm_blanking(), setup_hspwm_trigger(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock() setup_hspwm(), setup_hspwm_secondary()

## set_hspwm_duty_adjustment( )

**Syntax:**
set_hspwm_duty_adjustment(**unit, value**);

**Parameters:**
***unit*** - The High-Speed PWM unit to set.

***value*** - An int8 value to set the PWM unit's duty cycle adjustment value to.

**Returns:**
---

**Function:**
To setup the High-Speed PWM (HSPWM) duty cycle adjustment register. This is the value that is added to the duty cycle when the PCI source is active.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
set_hspwm_duty_ajustment(1, 10);
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## set_hspwm_override( )

**Syntax:**
set_hspwm_override(**unit, setting**);

**Parameters:**
**unit** - the High Speed PWM unit to override.

**settings** - the override settings to use. The valid options vary depending on the device. See the device's .h file for all options. Some typical options include:
HSPWM_FORCE_H_1
HSPWM_FORCE_H_0

HSPWM_FORCE_L_1
HSPWM_FORCE_L_0

**Returns:**
Undefined

**Function:**
Setup and High Speed PWM override settings.

**Availability:**
Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**
---

**Examples:**

```
setup_hspwm_override(1,HSPWM_FORCE_H_1|HSPWM_FORCE_L_0);
```

**See Also:**
setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_blanking(), setup_hspwm_trigger(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock(), setup_hspwm(), setup_hspwm_secondary()

## set_hspwm_period( )

**Syntax:**
set_hspwm_period(**period**);
set_hspwm_period(**unit, value**);

**Parameters:**
***period*** - An int16 value to set the PWM master period to.

***unit*** - The High-Speed PWM unit to set.

***value*** - An int16 value to set the PWM unit's period to.

**Returns:**
---

**Function:**
Sets up the High-Speed PWM (HSPWM) period registers.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
  set_hspwm_period(0x8000);     //set master period
  set_hspwm_period(4,0x9000);   //set PWM unit 4 period
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## set_hspwm_phase( )

**Syntax:**
set_hspwm_phase(**unit, primary, [secondary]**);

**Parameters:**
**unit** - The High Speed PWM unit to set.

**primary** - A 16-bit constant or variable to set the primary duty cycle.

**secondary** - An optional 16-bit constant or variable to set the secondary duty cycle. Secondary duty cycle is only used in Independent PWM mode. Not available on all devices, refer to device datasheet for availability.

**Returns:**
Undefined

**Function:**
Sets up the specified High Speed PWM unit.

**Availability:**
Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
set_hspwm(1,0x1000,0x8000);
```

**See Also:**
setup_hspwm_unit(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_blanking(), setup_hspwm_trigger(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock() setup_hspwm(), setup_hspwm_secondary()

## set_hspwm_scaling( )

**Syntax:**
set_hspwm_scaling(**period, inc_value**);

**Parameters:**
***period*** - An int16 value to set the frequency scaling minimum period to.

***inc_value*** - An int16 value to set the value added to the frequency scaling accumulator for each PWM clock.

**Returns:**
---

**Function:**
To setup the High-Speed PWM (HSPWM) frequency scaling registers.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**

```
set_hspwm_scaling(0x8000, 16);
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(),hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## set_hspwm_scaling( )

**Syntax:**
set_hspwm_trigger_a(**unit, value**);
set_hspwm_trigger_b(**unit, value**);
set_hspwm_trigger_c(**unit, value**);

**Parameters:**
***unit*** - The High-Speed PWM unit to set.

***value*** - An int16 value to set the PWM unit's trigger *x* register to.

**Returns:**
---

**Function:**
To setup the High-Speed PWM (HSPWM) trigger A, B and C registers.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
set_hspwm_trigger_a(1, 0x1000);
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(),

setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## set_input_level_x( )

**Syntax:**
set_input_level_a(**value**)
set_input_level_b(**value**)
set_input_level_v(**value**)
set_input_level_d(**value**)
set_input_level_e(**value**)
set_input_level_f(**value**)
set_input_level_g(**value**)
set_input_level_h(**value**)
set_input_level_j(**value**)
set_input_level_k(**value**)
set_input_level_l(**value**)

**Parameters:**
**value-** is an 8-bit int with each bit representing a bit of the I/O port.

**Returns:**
Undefined

**Function:**
These functions allow the I/O port Input Level Control (INLVLx) registers to be set. Each bit in the value represents one pin. A 1 sets the corresponding pin's input level to Schmitt Trigger (ST) level, and a 0 sets the corresponding pin's input level to TTL level.

**Availability:**
All devices with ODC registers, however not all devices have all I/O ports and not all devices port's have a corresponding ODC register.

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
set_input_level_a(0x0);   //sets PIN_A0 input level to ST and
all other
```

```
                                //PORTA pins to TTL level
```

**See Also:**
output_high(), output_low(), output_bit(), output_x(), General Purpose I/O

## set_motor_pwm_duty( )

**Syntax:**
set_motor_pwm_duty(***pwm***,***group***,***time***);

**Parameters:**
***pwm***- Defines the pwm module used.

***group***- Output pair number 1,2 or 3.

***time***- The value set in the duty cycle register.

**Returns:**
Void

**Function:**
Configures the motor control PWM unit duty.

**Availability:**
Devices that have the motor control PWM unit.

**Requires:**
---

**Examples:**

```
set_input_level_a(0x0);   //sets PIN_A0 input level to ST and
all other
                          //PORTA pins to TTL level
```

**See Also:**
get_motor_pwm_count(), set_motor_pwm_event(), set_motor_unit(), setup_motor_pwm()

## set_motor_pwm_event( )

**Syntax:**
set_motor_pwm_event(pwm,time);
*[PCD]* set_motor_pwm_event(pwm,time,[postscale]);

**Parameters:**
***pwm***- Defines the pwm module used.

***time***- The value in the special event comparator register used for scheduling other events.

*[PCD]* **postscale-** Optional parameter to set the special trigger output postscale (1-16). Defaults to 1 if not specified.

**Returns:**
Void

**Function:**
Configures the PWM event on the motor control unit.

**Availability:**
Devices that have the motor control PWM unit.

**Requires:**
---

**Examples:**

```
set_motor_pww_event(pwm,time);
```

```
[PCD] set_motor_pwm_event(1,625,2);
```

**See Also:**
get_motor_pwm_count(), setup_motor_pwm(), set_motor_unit(), set_motor_pwm_duty()

## set_motor_unit( )

**Syntax:**
set_motor_unit(***pwm***,***unit***,***options***, ***active_deadtime***, ***inactive_deadtime***);

**Parameters:**
***pwm***- Defines the pwm module used

***Unit***- This will select Unit A or Unit B

***options***- The mode of the power PWM module. See the devices .h file for all options

***active_deadtime***- Set the active deadtime for the unit

***inactive_deadtime***- Set the inactive deadtime for the unit

**Returns:**
Void

**Function:**
Configures the motor control PWM unit.

**Availability:**
Devices that have the motor control PWM unit.

**Requires:**
---

**Examples:**

```
set_motor_unit(pwm,unit,MPWM_INDEPENDENT | MPWM_FORCE_L_1,
active_deadtime,
inactive_deadtime);
```

**See Also:**
get_motor_pwm_count(), set_motor_pwm_event(), set_motor_pwm_duty(), setup_motor_pwm()

## set_nco_accumulator( )

**Syntax:**
set_nco_accumulator(**value**);

**Parameters:**
**value** - The 20-bit value to set the NCO accumulator to.

**Returns:**
---

**Function:**
Used to set the NCO accumulator to a specific value.

**Availability:**
Devices with a Numerically Controlled Oscillator (NCO) module.

**Requires:**
---

**Examples:**

```
set_nco_accumulator(500000);
```

**See Also:**
setup_nco( ), get_nco_accumulator( ), set_nco_inc_value( ), get_nco_inc_value( )

## set_nco_inc_value( )

**Syntax:**
set_nco_inc_value**(value**);

**Parameters:**
**value-** *value to set the NCO increment registers*

**Returns:**
Undefined

**Function:**
Sets the value that the NCO's accumulator will be incremented by on each clock pulse. The increment registers are double buffered so the new value won't be applied until the accumulator rolls-over.

**Availability:**
Devices with a NCO module

**Requires:**
---

**Examples:**

```
set_nco_inc_value(inc_value);       //sets the new increment
value
```

**See Also:**
setup_nco( ), get_nco_accumulator( ), get_nco_inc_value( )

## set_open_drain_x(value)

**Syntax:**
set_open_drain_a(value)
set_open_drain_b(value)
set_open_drain_c(value)
set_open_drain_d(value)

set_open_drain_e(value)
set_open_drain_f(value)
set_open_drain_g(value)
set_open_drain_h(value)
set_open_drain_j(value)
set_open_drain_k(value)

**Parameters:**
**value** – is an 8-bit int with each bit representing a bit of the I/O port.
*[PCD]* **value** – is a 16-bit int with each bit representing a bit of the I/O port.

**Returns:**
---

**Function:**
These functions allow the I/O port Open-Drain Control (ODCONx) registers to be set. Each bit in the value represents one pin. A 1 sets the corresponding pin to act as an open-drain output, and a 0 sets the corresponding pin to act as a digital output.
*[PCD]* Enables/Disables open-drain output capability on port pins. Not all ports or port pins have open-drain capability, refer to devices data sheet for port and pin availability.

**Availability:**
Devices with a NCO module

**Requires:**
---

**Examples:**

```
set_open_drain_a(0x01);      //makes PIN_A0 an open-drain output.
set_open_drain_b(0x001);     //enables open-drain output on PIN-B0
                             //disable on all other port B pins
```

**See Also:**
output_high(), output_low(), output_bit(), output_x(), General Purpose I/O

## set_power_pwm_override( )

**Syntax:**
set_power_pwm_override(***pwm***, ***override***, ***value***)

**Parameters:**
***pwm -*** is a constant between 0 and 7
***Override -*** is true or false
***Value -*** is 0 or 1

**Returns:**
Undefined

**Function:**
***pwm -*** selects which module will be affected.

**Override -** determines whether the output is to be determined by the OVDCONS register or the PDC registers. When override is false, the PDC registers determine the output. When override is true, the output is determined by the value stored in OVDCONS.

***value -*** determines if pin is driven to it's active staet or if pin will be inactive. I will be driven to its active state, 0 pin will be inactive.

**Availability:**
All devices equipped with PWM.

**Requires:**
---

**Examples:**

```
set_power_pwm_override(1, true, 1);  //PWM1 will be overridden to
active state
set_power_pwm_override(1, false, 0); //PMW1 will not be overidden
```

**See Also:**
setup_power_pwm(), setup_power_pwm_pins(), set_power_pwmX_duty()

## set_power_pwmx_duty( )

**Syntax:**
set_power_pwm*X*_duty(***duty***)

**Parameters:**
***X*** is 0, 2, 4, or 6

***Duty*** is an integer between 0 and 16383

**Returns:**
Undefined

**Function:**
Stores the value of duty into the appropriate PDCXL/H register. This duty value is the amount of time that the PWM output is in the active state.

**Availability:**
All devices equipped with PWM.

**Requires:**
---

**Examples:**

```
set_power_pwmx_duty(4000);
```

**See Also:**
setup_power_pwm(), setup_power_pwm_pins(), set_power_pwm_override()

## set_pulldown( )

**Syntax:**
set_Pulldown(state [, pin])

**Parameters:**
**Pins** are defined in the devices .h file. If no pin is provided in the function call, then all of the pins are set to the passed in state.

**State** is either true or false.

**Returns:**
Undefined

**Function:**
Sets the pin's pull down state to the passed in state value. If no pin is included in the function call, then all valid pins are set to the passed in state.

**Availability:**
All devices equipped with pull-down hardware

**Requires:**
Pin constants are defined in the devices .h file

**Examples:**

```
set_pulldown(true, PIN_B0);     //Sets pin B0's pull down state to true
    set_pullup(false);          //Sets all pin's pull down state to
false
```

## set_pullup( )

**Syntax:**
set_pullup(**state**, [ **pin**])

**Parameters:**
Pins are defined in the devices .h file. If no pin is provided in the function call, then all of the pins are set to the passed in state.

**State** is either true or false.

Pins are defined in the devices .h file. The actual number is a bit address. For example, port a (byte 5 ) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #DEFINE PIN_A3 43 . The pin could also be a variable that has a value equal to one of the predefined pin constants. Note if no pin is provided in the function call, then all of the pins are set to the passed in state.

**Returns:**
Undefined

**Function:**
Sets the pin's pull up state to the passed in state value. If no pin is included in the function call, then all valid pins are set to the passed in state.

**Availability:**
All Devices

**Requires:**
Pin constants are defined in the devices .h file

**Examples:**

```
set_pullup(true, PIN_B0);       //Sets pin B0's pull up state to true

   set_pullup(false);          //Sets all pin's pull up state to false
```

## set_pwm1_duty( ) set_pwm2_duty( ) set_pwm3_duty( ) set_pwm4_duty( ) set_pwm5_duty( )

**Syntax:**
set_pwm1_duty (***value***)
set_pwm2_duty (***value***)
set_pwm3_duty (***value***)

set_pwm4_duty (***value***)
set_pwm5_duty (***value***)
*[PCD]* set_pwm*X*_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 most significant bits are not required. The 10 bit value is then used to determine the duty cycle of the PWM signal as follows:

duty cycle = value / [ 4 * (PR2 +1 ) ]

If an 8-bit value is used, the duty cycle of the PWM signal is determined as follows:

duty cycle=value/(PR2+1)

Where PR2 is the maximum value timer 2 will count to before toggling the output pin.

*[PCD]* PIC24FxxKLxxx devices, writes the 10-bit value to the PWM to set the duty. An 8-bit value may be used if the most significant bits are not required. The 10-bit value is then used to determine the duty cycle of the PWM signal as follows:

duty cycle = value / [ 4 * (PRx +1 ) ]

Where PRx is the maximum value timer 2 or 4 will count to before rolling over.

PIC24FxxKMxxx devices, wires the 16-bit value to the PWM to set the duty. The 16-bit value is then used to determine the duty cycle of the PWM signal as follows:

duty cycle=value/(CCPxPRL+1)

Where CCPxPRL is the maximum value timer 2 will count to before toggling the output pin.

**Availability:**
This function is only available on devices with CCP/PWM hardware.
*[PCD]* This function is only available on devices with MCCP and/or SCCP modules.

**Requires:**
---
**Examples:**

```
                                        // For a 20 mhz clock, 1.2 khz
  frequency,
                                        // t2DIV set to 16, PR2 set to 200
                                       // the following sets the duty to
  50% (or 416 us).
```

```
  long duty;

  duty = 408; // [408/(4*(200+1))]=0.5=50%
  set_pwm1_duty(duty);
```

[PIC24FxxKLxxx Devices]

```
                                        // 32 MHz clock
unsigned int16 duty;
setup_timer2(T2_DIV_BY_4, 199, 1);     //period=50us
setup_ccp1(CCP_PWM);

duty=400;
                              //duty=400/[4*(199+1)]=0.5=50%
set_pwm1_duty(duty);
```

[PIC24FxxKMxxx Devices]

```
                                        // 32 MHz clock
unsigned int16 duty;

setup_ccp1(CCP_PWM);
set_timer_period_ccp1(799);           //period=50us
duty=400;                             //duty=400/(799+1)=0.5=50%
set_pwm1_duty(duty);
```

**Example Files:**
ex_pwm.c

**See Also:**
setup_ccpX(), set_ccpX_compare_time(), set_timer_period_ccpX(), set_timer_ccpX(), get_timer_ccpX(), get_capture_ccpX(, get_captures32_ccpX()

## set_pwm1_offset( ) set_pwm2_offset( ) set_pwm3_offset( ) set_pwm4_offset( ) set_pwm5_offset( ) set_pwm6_offset( )

**Syntax:**
set_pwm1_offset (***value***)
set_pwm2_offset (***value***)
set_pwm3_offset (***value***)
set_pwm4_offset (***value***)
set_pwm5_offset (***value***)
set_pwm6_offset (***value***)

**Parameters:**
***value*** - 16-bit constant or variable

**Returns:**
Undefined

**Function:**
Writes the 16-bit to the PWM to set the offset. The offset is used to adjust the waveform of a slae PWM module relative to the waveform of a master PWM module.

**Availability:**
Devices with a 16-bit PWM module

**Requires:**
---

**Examples:**

```
set_pwm1_offset(0x0100);
set_pwm1_offset(offset);
```

**See Also:**
setup_pwm(), set_pwm_duty(), set_pwm_period(), clear_pwm_interrupt(), set_pwm_phase(), enable_pwm_interrupt(), disable_pwm_interrupt(), pwm_interrupt_active()

## set_pwm1_period( ) set_pwm2_period( ) set_pwm3_period( ) set_pwm4_period( ) set_pwm5_period( ) set_pwm6_period( )

**Syntax:**
set_pwm1_period (***value***)
set_pwm2_period (***value***)
set_pwm3_period (***value***)
set_pwm4_period (***value***)
set_pwm5_period (***value***)
set_pwm6_period (***value***)

**Parameters:**
***value -*** *16-bit constant or variable*

**Returns:**
Undefined

**Function:**
Writes the 16-bit to the PWM to set the period. When the PWM module is set-up for standard mode it sets the period of the PWM signal. When set-up for set on match mode, it sets the maximum value at which the phase match can occur. When in toggle on match and center aligned modes it sets the maximum value the PWMxTMR will count to, the actual period of PWM signal will be twice what the period was set to.

**Availability:**
Devices with a 16-bit PWM module

**Requires:**
---

**Examples:**

```
set_pwm1_period(0x8000);
set_pwm1_period(period);
```

**See Also:**
setup_pwm(), set_pwm_duty(), set_pwm_phase(), clear_pwm_interrupt(), set_pwm_offset(), enable_pwm_interrupt(), disable_pwm_interrupt(), pwm_interrupt_active()

## set_pwmx_phase( )

**Syntax:**
set_pwm1_phase (***value***)
set_pwm2_phase (***value***)
set_pwm3_phase (***value***)
set_pwm4_phase (***value***)
set_pwm5_phase (***value***)
set_pwm6_phase (***value***)

**Parameters:**
***value -*** *16-bit constant or variable*

**Returns:**
Undefined

**Function:**
Writes the 16-bit to the PWM to set the phase. When the PWM module is set-up for standard mode the phaes specifies the start of the duty cycle, when in set on match

mode it specifies when the output goes high, and when in toggle on match mode it specifies when the output toggles. Phase is not used when in center aligned mode.

**Availability:**
Devices with a 16-bit PWM module

**Requires:**
---

**Examples:**

```
set_pwm1_phase(0);
set_pwm1_phase(phase);
```

**See Also:**
setup_pwm(), set_pwm_duty(), set_pwm_period(), clear_pwm_interrupt(), set_pwm_offset(), enable_pwm_interrupt(), disable_pwm_interrupt(), pwm_interrupt_active()

## set_timerx( ) set_rtcc( ) set_timer0( ) set_timer1( ) set_timer2( ) set_timer3( ) set_timer4( ) set_timer5( )

**Syntax:**
set_timerX(***value***)
set_timer0(value) or set_rtcc (value)
set_timer1(value)
set_timer2(value)
set_timer3(value)
set_timer4(value)
set_timer5(value)

**Parameters:**
Timers 1 & 5 get a 16 bit int.
Timer 2 and 4 gets an 8 bit int.
Timer 0 (AKA RTCC) gets an 8 bit int except on the PIC18XXX where it needs a 16 bit int.
Timer 3 is 8 bit on PIC16 and 16 bit on PIC18

**Returns:**
Undefined

**Function:**
Sets the count value of a real time clock/counter. RTCC and Timer0 are the same. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2...)

**Availability:**
Timer 0 - All devices
Timers 1 & 2 - Most but not all PCM devices
Timer 3 - Only PIC18XXX and some pick devices
Timer 4 - Some PCH devices
Timer 5 - Only PIC18XX31

**Requires:**
---

**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

**See Also:**
set_timer1(), get_timerX() Timer0 Overview, Timer1Overview, Timer2 Overview, Timer5 Overview

## set_ticks( )

**Syntax:**
set_ticks([**stream**],**value**);

**Parameters:**
**stream** – optional parameter specifying the stream defined in #USE TIMER

**value** – a 8, 16 or 32 bit integer, specifying the new value of the tick timer. (int8, int16 or int32)
*[PCD]* **value** – a 8, 16, 32 or 64 bit integer, specifying the new value of the tick timer. (int8, int16, int32 or int64)

**Returns:**
Void

**Function:**
Sets the new value of the tick timer. Size passed depends on the size of the tick timer.

**Availability:**
All Devices

**Requires:**
#USE TIMER(options)

**Examples:**

```
#USE TIMER(TIMER=1,TICK=1ms,BITS=16,NOISR)

void main(void) {
   unsigned int16 value = 0x1000;

   set_ticks(value);
}                                          // 256-
(.000035/(4/20000000))
```

**See Also:**
#USE TIMER, get_ticks()

## setup_sd_adc_calibration( )

**Syntax:**
setup_sd_adc_calibration(model);

**Parameters:**
***mode***- selects whether to enable or disable calibration mode for the SD ADC module. The following defines are made in the device's *.h* file:
SDADC_START_CALIBRATION_MODE
SDADC_END_CALIBRATION_MODE

**Returns:**
---

**Function:**
To enable or disable calibration mode on the Sigma-Delta Analog to Digital Converter (SD ADC) module. This can be used to determine the offset error of the module, which then can be subtracted from future readings.

**Availability:**
Devices with a SD ADC module

**Requires:**
#USE TIMER(options)

**Examples:**

```
signed int 32 result, calibration;

set_sd_adc_calibration(SDADC_START_CALIBRATION_MODE);
calibration=read_sd_adc()
set_sd_adc_calibration(SDADC_END_CALIBRATION_MODE);

result=read_sd_adc()-calibration;
```

**See Also:**
setup_sd_adc(), read_sd_adc(), set_sd_adc_channel()

## set_sd_adc_channel( )

**Syntax:**
setup_sd_adc(channel);

**Parameters:**
***channel***- sets the SD ADC channel to read. Channel can be 0 to read the difference between CH0+ and CH0-, 1 to read the difference between CH1+ and CH1-, or one of the following:
SDADC_CH1SE_SVSS
SDADC_REFERENCE

**Returns:**
Void

**Function:**
To select the channel that the Sigma-Delta Analog to Digital Converter (SD ADC) performs the conversion on.

**Availability:**
Devices with a SD ADC module

**Requires:**
---

**Examples:**

```
set_sd_adc_channel(0);
```

**See Also:**
setup_sd_adc(), read_sd_adc(), set_sd_adc_calibration()

## set_slow_slew_x()

**Syntax:**
set_slow_slew_a(**value**);
set_slow_slew_b(**value**);
set_slow_slew_c(**value**);
set_slow_slew_d(**value**);
set_slow_slew_e(**value**);
set_slow_slew_f(**value**);
set_slow_slew_g(**value**);
set_slow_slew_h(**value**);

**Parameters:**
**value** - may be a 1-bit constant or an 8-bit value (see the device's header file to determine which) used to enable and disable slew rating limiting on a port or port pin. Devices that take a 1-bit constant passing a 1 to function enables slew rate limiting on entire port. Passing a 0 to function disables slew rate limiting on entire port. Devices that take an 8-bit value, each bit corresponds to a pin on the port. Setting a bit enables slew rate limiting on that port's corresponding pin and clearing a bit disables slew rate limiting on that port's corresponding pin.

**Returns:**
---

**Function:**
Used to enable and disable slew rate limiting on the device's ports or port pins.

**Availability:**
Devices that have Slew Rate Control registers for enabling and disabling slew rate limiting.

**Requires:**
---

**Examples:**

```
set_slow_slew_a(TRUE);
set_slow_slew_a(0x03);
```

**See Also:**
set_tris_x(), set_input_level_x(), set_open_drain_x(), get_tris_x(), output_x(), input_x(), input_change_x(), port_x_pullups(), input(), input_state(), output_low(), output_high(), output_toggle(), output_bit(), output_float(), output_drive()

## set_timerA( )

**Syntax:**
set_timerA(**value**);

**Parameters:**
An 8 bit integer. Specifying the new value of the timer. (int8)

**Returns:**
---

**Function:**
Sets the current value of the timer. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2, …)

**Availability:**
Devices with Timer A hardware

**Requires:**
---

**Examples:**

```
                                    // 20 mhz clock, no prescaler, set
  timer A
                                    // to overflow in 35us
  set_timerA(81);                  // 256-(.000035/(4/20000000)
```

**See Also:**
get_timerA( ), setup_timer_A( ), TimerA Overview

## set_timerB( )

**Syntax:**
set_timerB(**value**);

**Parameters:**
An 8 bit integer. Specifying the new value of the timer. (int8)

**Returns:**
---

**Function:**
Sets the current value of the timer. All timers count up. When a timer reaches the maximum value it will flip over to 0 and continue counting (254, 255, 0, 1, 2, …)

**Availability:**
Devices with Timer B hardware

**Requires:**
---

**Examples:**

```
                                    // 20 mhz clock, no prescaler, set
  timer B
                                    // to overflow in 35us
  set_timerB(81);                  // 256-(.000035/(4/20000000)
```

**See Also:**
get_timerB( ), setup_timer_B( ), TimerB Overview

## set_timerxy( )

**Syntax:**
set_timerXY(***value***)

**Parameters:**
A 32 bit integer, specifying the new value of the timer. (int32)

**Returns:**
---

**Function:**
Retrieves the 32 bit value of the timers X and Y, specified by XY(which may be 23, 45, 67 and 89)

**Availability:**
This function is available on all devices that have a valid 32 bit enabled timers. Timers 2 & 3, 4 & 5, 6 & 7 and 8 & 9 may be used. The target device must have one of these timer sets. The target timers must be enabled as 32 bit.

**Requires:**
---

**Examples:**

```
if(get_timer45() == THRESHOLD)
   set_timer(THRESHOLD + 0x1000);              //skip those timer
values
```

**See Also:**
Timer Overview, setup_timerX(), get_timerXY(), set_timerX(), set_timerXY()

## set_timer_ccp1( ) set_timer_ccp2( ) set_timer_ccp3( ) set_timer_ccp4( ) set_timer_ccp5( ) set_timer_ccp6( )

**Syntax:**
set_timer_ccpx(**time**);
set_timer_ccpx(**timeL, timeH**);

**Parameters:**
**time** - may be a 32-bit constant or variable. Sets the timer value for the CCPx module when in 32-bit mode.

**timeL** - may be a 16-bit constant or variable to set the value of the lower timer when CCP module is set for 16-bit mode.

**timeH** - may be a 16-bit constant or variable to set the value of the upper timer when CCP module is set for 16-bit mode.

**Returns:**
---

**Function:**
This function sets the timer values for the CCP module. TimeH is optional parameter when using 16-bit mode, defaults to zero if not specified.

**Availability:**
Available only on PIC24FxxKMxxx family of devices with a MCCP and/or SCCP modules.

**Requires:**
---

**Examples:**

```
setup_ccp1(CCP_TIMER);      //set for dual timer mode
set_timer_ccp1(100,200);    //set lower timer value to 100 and upper
timer
                            //value to 200
```

**See Also:**
set_pwmX_duty(), setup_ccpX(), set_ccpX_compare_time(), get_capture_ccpX(), set_timer_period_ccpX(), get_timer_ccpx(), get_captures32_ccpX()

## set_timer_period_ccp1( ) set_timer_period_ccp2( ) set_timer_period_ccp3( ) set_timer_period_ccp4( ) set_timer_period_ccp5( ) set_timer_period_ccp6( )

**Syntax:**
set_timer_period_ccpx(**time**);
set_timer_period_ccpx(**timeL, timeH**);

**Parameters:**
**time** - may be a 32-bit constant or variable. Sets the timer value for the CCPx module when in 32-bit mode.

**timeL** - may be a 16-bit constant or variable to set the value of the lower timer when CCP module is set for 16-bit mode.

**timeH** - may be a 16-bit constant or variable to set the value of the upper timer when CCP module is set for 16-bit mode.

**Returns:**
---

**Function:**
This function sets the timer periods for the CCP module. When setting up CCP module in 32-bit function is only needed when using Timer mode. Period register are not used when module is setup for 32-bit compare mode, period is always 0xFFFFFFFF. TimeH is optional parameter when using 16-bit mode, default to zero if not specified.

**Availability:**
Available only on PIC24FxxKMxxx family of devices with a MCCP and/or SCCP modules.

**Requires:**
---

**Examples:**

```
setup_ccp1(CCP_TIMER);              //set for dual timer mode
set_timer_period_ccp1(800,2000);    //set lower timer period to 800 and
                                    //upper timer period to 2000
```

**See Also:**
set_pwmX_duty(), setup_ccpX(), set_ccpX_compare_time(), set_timer_ccpX(), get_timer_ccpX(), get_capture_ccpX(), get_captures32_ccpX()

## set_tris( )

**Syntax:**
set_tris_a (***value***)
set_tris_b (***value***)
set_tris_c (***value***)
set_tris_d (***value***)
set_tris_e (***value***)
set_tris_f (***value***)
set_tris_g (***value***)
set_tris_h (***value***)
set_tris_j (***value***)
set_tris_k (***value***)
set_tris_l (***value***)

**Parameters:**
***value*** is an **8** bit int with each bit representing a bit of the I/O port.
*[PCD]* ***value*** is an **16** bit int with each bit representing a bit of the I/O port.

**Returns:**
Undefined

**Function:**
These functions allow the I/O port direction (TRI-State) registers to be set.
This must be used with FAST_IO and when I/O ports are accessed as memory such as when a # BYTE directive is used to access an I/O port.
*[PCD]* This must be used with FAST_IO and when I/O ports are accessed as memory such as when a #word directive is used to access an I/O port.
Using the default standard I/O the built in functions set the I/O direction automatically.
Each bit in the value represents one pin. A 1 indicates the pin is input and a 0 indicates it is output.

**Availability:**
All devices (however not all devices have all I/O ports)

**Requires:**
Pin constants are defined in the devices .h file

**Examples:**

```
SET_TRIS_B( 0x0F );        // B7,B6,B5,B4 are outputs
                           //  B3,B2,B1,B0 are inputs
                     [PCD] // B15,B14,B13,B12,B11,B10,B9,B8,
```

**Example Files:**
lcd.c

**See Also:**
#USE FAST_IO, #USE FIXED_IO, #USE STANDARD_IO, General Purpose I/O

## set_uart_speed( )

**Syntax:**
set_uart_speed (***baud***, [***stream, clock***])

**Parameters:**
**baud -** is a constant representing the number of bits per second.

**stream -** is an optional stream identifier.

**clock -** is an optional parameter to indicate what the current clock is if it is different from the #use delay value

**Returns:**
---

**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

**See Also:**
#USE RS232, putc(), getc(), setup uart(), RS232 I/O Overview

## setjmp( )

**Syntax:**
result = setjmp (***env***)

**Parameters:**
**env -** The data object that will receive the current environment

**Returns:**
If the return is from a direct invocation, this function returns 0.
If the return is from a call to the longjmp function, the setjmp function returns a nonzero value and it's the same value passed to the longjmp function.

**Function:**
Stores information on the current calling context in a data object of type jmp_buf and which marks where you want control to pass on a corresponding longjmp call.

**Availability:**
All Devices

**Requires:**
#INCLUDE <setjmp.h>

**Examples:**

```
result = setjmp(jmpbuf);
```

**See Also:**
longjmp()

## setup_act( )

**Syntax:**
setup_act(**settings**);

**Parameters:**
**settings** - to setup the ACT module. See the device's header file for options.

**Returns:**
---

**Function:**
Used to setup the Active Clock Tuning (ACT) module.

**Availability:**
Devices with an ACT module. See the device's header file for availability.

**Requires:**
---

**Examples:**

```
setup_act(ACT_ENABLED | ACT_TUNED_TO_USE);
```

**See Also:**
act_status( )

## setup_adc(mode)

## [PCD] setup_adc2(mode)

**Syntax:**
setup_adc (***mode, [ADCRS], [ADRPT]***);
*[PCD]* setup_adc2(***mode***);

**Parameters:**
***mode***- Analog to digital mode. The valid options vary depending on the device. See the devices .h file for all options. Some typical options include:

ADC_OFF
ADC_CLOCK_INTERNAL
ADC_CLOCK_DIV_32
*[PCD]* ADC_CLOCK_INTERNAL – The ADC will use an internal clock
*[PCD]* ADC_CLOCK_DIV_32 – The ADC will use the external clock scaled down by 32
*[PCD]* ADC_TAD_MUL_16 – The ADC sample time will be 16 times the ADC conversion time

***ADCRS*** - For devices with an analog-to-digital converter with computation (ADC2) module only. Optional parameter used set how much the accumulated value is divided by (2^ADCRS) in Accumulate, Average and Parst Average modes, and the cut-off frequency in low-pass filter mode.

***ADRPT*** - For devices with an ADC2 module only. Optional parameter used to set the number of samples to be done before performing a threshold comparison in Average, Bust Average and low-pass filter modes.

**Returns:**
---

**Function:**
Configures the analog to digital converter.
*[PCD]* Configures the ADC clock speed and the ADC sample time. The ADC converters have a maximum speed of operation, so ADC clock needs to be scaled accordingly. In addition, the sample time can be set by using a bitwise OR to concatenate the constant to the argument.

**Availability:**
Only the devices with built in analog to digital converter.

**Requires:**
Constants are defined in the devices .h file.

**Examples:**

```
setup_adc_ports( ALL_ANALOG );
setup_adc(ADC_CLOCK_INTERNAL );
set_adc_channel( 0 );
value = read_adc();
setup_adc( ADC_OFF )
```

**Example Files:**
ex_admm.c

**See Also:**
setup_adc_ports(), set_adc_channel(), read_adc(), #DEVICE, ADC Overview

## setup_adc_ports( )

## [PCD] setup_adc_ports2( )

**Syntax:**
setup_adc_ports (**value**)
setup_adc_ports (**ports**, [**reference**])
setup_adc_ports (**ports, [ports2], [reference]**)
setup_adc_ports (**ports, [p2_ports], [reference]**)
*[PCD]* setup_adc_ports (**ports**, **reference**])

**Parameters:**
***value*** - a constant defined in the device's .h file

***ports*** - a constant specifying the ADC pins to use.

***ports2*** - an optional constant on devices with more than 32 analog pins. To specify the ADC pins to use for analog pins 32-x.

***p2_ports*** - an optional constant specifying the ADC pins to use on a device that have more than 32 analog pins. Not an option on all devices, see the device's header file to determine if available. For devices with this option, if setting ADC reference, the reference is passed as the third parameter.

***reference*** - is an optional constant specifying the ADC reference to use. By default, the reference voltage are Vss and Vdd

**Returns:**
---

**Function:**
Sets up the ADC pins to be analog, digital, or a combination and the voltage reference to use when computing the ADC value. The allowed analog pin combinations vary depending on the chip and are defined by using the bitwise OR to concatenate selected pins together. Check the device include file for a complete list of available pins and reference voltage settings. The constants ALL_ANALOG and NO_ANALOGS are valid for all chips.
Some other example pin definitions are:

sAN1 | sAN2 - AN1 and AN2 are analog, remaining pins are digital
sAN0 | sAN3 - AN0 and AN3 are analog, remaining pins are digital

**Availability:**
This function is only available on devices with A/D hardware.
This function is only available on devices with built-in A/D converters

**Requires:**
Constants are defined in the devices .h file.

**Examples:**

```
  setup_adc_ports(ALL_ANALOG);              // All pins analog (that can
  be)

  setup_adc_ports(RA0_RA1_ANALOG_RA3_REF); // Pins A0 and A1 are analog.
   Pin RA3 is
                                           // used for the reference
  voltage and all
                                           // other pins are digital.
  setup_adc_ports(ALL_ANALOG);             // Set all ADC pins to analog
  mode.

  setup_adc_ports(sAN0|sAN1|sAN3);        // Pins AN0, AN1 and AN3 are
  analog and all
                                          // others pins are digital.
  setup_adc_ports(sAN0|sAN1, VREF_VDD); // Pins AN0 and AN1 are analog.
   The VrefL pin
                                         // and Vdd are used for voltage
references.
```

**Example Files:**
ex_admm.c

**See Also:**
setup_adc_reference(), set_adc_channel(), read_adc(), setup_adc(), set_analog_pins(), ADC Overview

## setup_adc_reference( ) setup_adc_reference2( )

**Syntax:**
**setup_adc_reference(reference)**

**Parameters:**
**reference -** the voltage reference to set the ADC. The valid options depend on the device, see the device's .h file for all options. Typical options include:
- VSS_VDD
- VSS_VREF
- VREF_VREF
- VREF_VDD

**Returns:**
Undefined.

**Function:**
Set the positive and negative voltage reference for the Analog to Digital Converter (ADC) uses.

**Availability:**
Only on devices with an ADC and has ANSELx, x being the port letter; registers for setting which pins are analog or digital.

**Requires:**
Nothing

**Examples:**
```
setup_adc_reference(VSS_VREF);
```

**Examples Files:**
None

**See Also:**
set_analog_pins(), set_adc_channel(), read_adc(), setup_adc(), setup_adc_ports(), ADC Overview

## setup_adc_reference( ) setup_adc_reference2( )

**Syntax:**
**setup_adc_reference(reference)**

**Parameters:**
**reference -** the voltage reference to set the ADC. The valid options depend on the device, see the device's .h file for all options. Typical options include:
- VSS_VDD
- VSS_VREF
- VREF_VREF
- VREF_VDD

**Returns:**
Undefined.

**Function:**
Set the positive and negative voltage reference for the Analog to Digital Converter (ADC) uses.

**Availability:**
Only on devices with an ADC and has ANSELx, x being the port letter; registers for setting which pins are analog or digital.

**Requires:**
Nothing

**Examples:**
```
setup_adc_reference(VSS_VREF);
```

**Examples Files:**
None

**See Also:**
set_analog_pins(), set_adc_channel(), read_adc(), setup_adc(), setup_adc_ports(), ADC Overview

## setup_at( )

**Syntax:**
setup_at(**settings)**

**Parameters:**
**settings** - the setup of the AT module. See the device's header file for all options. Some typical options include:
at_enabled
at_disabled
at_multi_pulse_mode
at_single_pulse_mode

**Returns:**
---

**Function:**
To setup the Angular Timer (AT) module.

**Availability:**
All devices with an AT module

**Requires:**
Constants defined in the device's .h file

**Examples:**
```
setup_at(AT_ENABLED|AT_MULTI_PULSE_MODE|AT_INPUT_ATIN);
```

**See Also:**
at_set_resolution(), at_get_resolution(), at_set_missing_pulse_delay(), at_get_missing_pulse_delay(), at_get_period(), at_get_phase_counter(), at_set_set_point(), at_get_set_point(), at_get_set_point_error(), at_enable_interrupts(), at_disable_interrupts(), at_clear_interrupts(), at_interrupt_active(), at_setup_cc(), at_set_compare_time(), at_get_capture(), at_get_status()

## setup_capture( )

**Syntax:**
setup_capture(***x***, ***mode***)

**Parameters:**
***x* -** is 1-16 and defines which input capture module is being configured

***mode* -** is defined by the constants in the devices .h file

**Returns:**
---

**Function:**
This function specifies how the input capture module is going to function based on the value of mode. The device specific options are listed in the device .h file

**Availability:**
Only available on devices with Input Capture modules

**Requires:**
---

**Examples:**
```
setup_timer3(TMR_INTERNAL | TMR_DIV_BY_8);
setup_capture(2, CAPTURE_FE | CAPTURE_TIMER3);
while(TRUE) {
      timerValue = get_capture(2, TRUE);
      printf("Capture 2 occurred at: %LU", timerValue);
}
```

**See Also:**
get_capture( ), setup_compare( ), Input Capture Overview

## setup_ccp1( ) setup_ccp2( ) setup_ccp3( ) setup_ccp4( ) setup_ccp5( ) setup_ccp6( ) setup_ccp8( ) setup_ccp9( ) setup_ccp10( )

**Syntax:**
setup_ccp1 (***mode***) or setup_ccp1 (***mode, pwm***)
setup_ccp2 (***mode***) or setup_ccp2 (***mode, pwm***)
setup_ccp3 (***mode***) or setup_ccp3 (***mode, pwm***)
setup_ccp5 (***mode***) or setup_ccp5 (***mode, pwm***)
setup_ccp6 (***mode***) or setup_ccp6 (***mode, pwm***)
*[PCD]* setup_ccpx(**mode**,[**pwm**]);//PIC24FxxKLxxx devices
*[PCD]* setup_ccpx(**mode1**,[**mode2**],[**mode3**],[**dead_time**]);//PIC24FxxKMxxx devices

**Parameters:**
***mode* -** is a constant. Valid constants are defined in the devices .h file and refer to devices .h file for all options; some options are as follows:

**Disable the CCP**

| | |
|---|---|
| CCP_CAPUTURE_FE | Capture on falling edge |
| CCP_CAPUTURE_RE | Capture on rising edge |
| CCP_CAPUTURE_DIV_4 | Capture after 4 pulses |
| CCP_CAPUTURE_DIV_16 | Capture after 16 pulses |

**Set CCP to Capture Mode:**

| | |
|---|---|
| CCP_CAPUTURE_SET_ON_MATCH | Output high on compare |
| CCP_CAPUTURE_CLR_ON_MATCH | Output low on compare |
| CCP_CAPUTURE_INT | Interrupt on compare |
| CCP_CAPUTURE_RESET_TIMER | Reset timer on compare |

**Set to CCP to PWM Mode:**

| | |
|---|---|
| CCP_PWM | Enable Pulse Width Modulator |

**Constants used for ECCP Modules:**
CCP_PWM_H_H
CCP_PWM_H_L
CCP_PWM_L_H
CCP_PWM_L_L

CCP_PWM_FULL_BRIDGE
CCP_PWM_FULL_BRIDGE_REV
CCP_PWM_HALF_BRIDGE

| | |
|---|---|
| CCP_SHUTDOWN_ON_COMP1 | Shutdown on Comparator 1 change |
| CCP_SHUTDOWN_ON_COMP2 | Shutdown on Comparator 2 change |
| CCP_SHUTDOWN_ON_COMP | Either Comparator 1 or 2 change |
| CCP_SHUTDOWN_ON_INTO | VIL on INT pin |
| CCP_SHUTDOWN_ON_COMP1_INTO | VIL on INT pin or Comparator 1 change |
| CCP_SHUTDOWN_ON_COMP2_INTO | VIL on INT pin or Comparator 2 change |
| CCP_SHUTDOWN_ON_COMP_INTO | VIL on INT pin or Comparator 1 or 2 change |

| | |
|---|---|
| CCP_SHUTDOWN_AC_L | Drive pins A and C high |
| CCP_SHUTDOWN_AC_H | Drive pins A and C low |
| CCP_SHUTDOWN_AC_F | Drive pins A and D tri-state |

| | |
|---|---|
| CCP_SHUTDOWN_BD_L | Drive pins B and D high |
| CCP_SHUTDOWN_BD_H | Drive pins B and D low |
| CCP_SHUTDOWN_BD_F | Drive pins B and D tri-state |

| | |
|---|---|
| CCP_SHUTDOWN_RESTART | Device restart after a shutdown event |
| CCP_DELAY | Use the deadband delay |

***pwm parameter -*** is an optional parameter for chips that includes ECCP module. This parameter allows setting the shutdown time. The value may be 0-255.

*[PCD]* ***mode*** and ***mode1 -*** constants used for setting up the CCP module. Valid constants are defined in the device's .h file; refer to the device's .h file for all options. Some typical options are as follows:

CCP_OFF
CCP_COMPARE_INT_AND_TOGGLE
CCP_COMPARE_FE
CCP_COMPARE_RE
CCP_COMPARE_DIV_4
CCP_COMPARE_DIV_16
CCP_COMPARE_SET_ON_MATCH
CCP_COMPARE_CLR_ON_MATCH
CCP_COMPARE_INT
CCP_COMPARE_RESET_TIMER
CCP_PWM

*[PCD]* **mode2** is an optional parameter for setting up more settings of the CCP module. Valid constants are defined in the device's .h file, refer to the device's .h file for all options.

*[PCD]* **mode3** is an optional parameter for setting up more settings of the CCP module. Valid constants are defined in the device's .h file, refer to the device's .h file for all options.

*[PCD]* **pwm** is an optional parameter for devices that have an ECCP module. this parameter allows setting the shutdown time. The value may be 0-255.

*[PCD]* **dead_time** is an optional parameter for setting the dead time when the CCP module is operating in PWM mode with complementary outputs. The value may be 0-63, 0 is the default setting if not specified.

**Returns:**
---

**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.

*[PCD]* Initializes the CCP module. For PIC24FxxKLxxx devices the CCP module can operate in three modes (Capture, Compare or PWM).
**Capture Mode** - the value of Timer 3 is copied to the CCPRxH and CCPRxl registers when an input event occurs.

**Compare Mode** - will trigger an action when Timer 3 and the CCPRxL and CCPRxH registers are equal.

**PWM Mode** - will generate a square wave, the duty cycle of the signal can be adjusted using the CCPRxL register and the DCxB bits of the CCPxCON register. The function set_pwmx_duty() is provided for setting the duty cycle when in PWM mode.

PIC24FxxKMxxx devices, the CCP module can operate in four mode (Timer, Caputure, Compare or PWM). IN Timer mode, it functions as a timer. The module has to basic modes, it can functions as two independent 16-bit timers/counters or as a single 32-bit timer/counter. The mode it operates in is controlled by the option CCP_TIMER_32_BIT, with the previous options added, the module operates as a single 32-bit timer, and if not added, it operates as two 16-bit timers. The function set_timer_period_ccpx() is provided to set the period(s) of the timer, and the functions set_timer_ccpx() and get_timer_ccpx() are provided to set and get the current value of the timer(s).

In Capture mode, the value of the timer is captured when an input event occurs, it can operate in either 16-bit or 32-bit mode. The functions get_capture_ccpx() and get_capture32_ccpx() are provided to get the last capture value.

In Compare and PWM modes, the value of the timers is c ompared to one or two compare registers, depending on its mode of operation, to generate a single output transition or a train of output pulses. For signal output edge modes, CCP_COMPARE_SET_ON_MATCH, CCP_COMPARE_CLR_ON_MATCH, and CCP_COMPARE_TOGGLE, the module can operate in 16 or 32-bit mode, all other modes can only operate in 16-bit mode. However, when in 32-bit mode the timer source will only rollover when it reaches 0xFFFFFFFF or when reset from an external synchronization source. Therefore, is a period of less than 0xFFFFFFFF is needed, as it requires an external synchronization source to reset the timer. The functions set_ccpx_compare_time() and set_pwmx_duty() are provided for setting the compare registers.

**Availability:**
This function is only available on devices with CCP hardware.
*[PCD]* Only on devices with the MCCP and/or SCCP modules.

**Requires:**
Constants are defined in the devices .h file.

**Examples:**
```
setup_ccp1(CCP_CAPTURE_RE);
[PCD]
setup_ccp1(CCP_CAPTURE_FE);
setup_ccp1(CCP_COMPARE_TOGGLE);
setup_ccp1(CCP_PWM);
```

**Example Files:**
ex_pwm.c, ex_ccpmp.c, ex_ccp1s.c

**See Also:**
set_pwmX_duty(), set_ccpX_compare_time(), set_timer_period_ccpX(), set_timer_ccpX(), get_timer_ccpX(), get_capture_ccpX(), get_captures32_ccpX()

## setup_clc1() setup_clc2() setup_clc3() setup_clc4()

**Syntax:**
setup_clc1**(mode);**
setup_clc2**(mode);**
setup_clc3**(mode);**
setup_clc4**(mode);**_capture(***x***, ***mode***)

**Parameters:**
**mode** – The mode to setup the Configurable Logic Cell (CLC) module into. See the device's .h file for all options. Some typical options include:

CLC_ENABLED
CLC_OUTPUT
CLC_MODE_AND_OR
CLC_MODE_OR_XOR

**Returns:**
---

**Function:**
Sets up the CLC module to performed the specified logic. Please refer to the device datasheet to determine what each input to the CLC module does for the select logic function

**Availability:**
Devices with a CLC module

**Requires:**
---

**Examples:**
```
setup_clc1(CLC_ENABLED | CLC_MODE_AND_OR);
```

**See Also:**
clcx_setup_gate(), clcx_setup_input()

## setup_comparator( )

**Syntax:**
setup_comparator (***mode***)
*[PCD]* setup_comparator (**comparator, *mode***);
*[PCD]* setup_comparator (**comparator, *mode***, [**blanking_period]**);

**Parameters:**
***mode*** is a constant. Valid constants are in the devices .h file refer to devices .h file for valid options. Some typical options 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

*[PCD]* **comparator** - constant specifying which comparator to setup.

*[PCD]* ***mode -*** constants specifying the settings to setup the specified comparator. See the device's .h file for all options. Some typical options include:

CXINB_CXINA
CXINC_CSINA
CXIND_CXINA
CXINB_VREF
CXINC_VREF
CXIND_VREF
COMP_INVERT
COMP_OUTPUT

*PCD]* **blanking_period -** optional parameter available on devices with an Analog Comparator with Slope Compensation DAC peripheral. See the device's header file for availability. It sets the 10-bit blanking period for the comparator following changes to the DAC output during Change-of-State.

**Returns:**
---

**Function:**
Sets the analog comparator module. The above constants have four parts representing the inputs: C1-, C1+, C2-, C2+

*[PCD]* Configures the voltage comparator. The voltage comparators allow to compare two voltages and find the greater of them. The configuration constants for this function specify the sources for the comparator in the order Cx- and Cx+. The results of the comparator modules are stored in CxOUT. COMP_INVERT will invert the result of the comparator and COMP_OUTPUT will output the result to the comparator output pin.

**Availability:**
This function is only available on devices with an analog comparator.
*[PCD]* Devices with a comparator module.

**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);
[PCD]  setup_comparator(1,CXINB_CXINA);    // setup C1
       setup_comparator(2,CXINB_CXINA);    // setup C2
```

**Example Files:**
ex_comp.c

**See Also:**
Analog Comparator Overview, setup_comparator_filter(), setup_comparator_mask(), setup_comparator_dac(), setup_comparator_slope()

## [PCD]setup_comparator_dac( )

**Syntax:**
[PCD] setup_comparator_dac(**settings**);
[PCD] setup_comparator_dac(**settings**, [**tmode_time], [ss_time])**;

**Parameters:**
[PCD] **settings** - constants specifying the settings to setup the comparator and DAC. See the device's header file for defines that can be used with this parameter.

[PCD] ***tmode_time* -** optional 10-bit value used the set the Transition Mode Duration, value passed for this parameter should be less the ss-time parameter.

PCD] **ss_time -** optional 10-bit parameter used to set the time from Start of Transmission Mode until Steady-State filter is enabled.

**Returns:**
---

**Function:**
[PCD] Used to setup the common settings for the Analog Comparator with Slope Compensation DAC peripheral. Some common settings include enabling/disabling the common DAC module, DAC clock source, comparator filter clock divider, etc.

**Availability:**
[PCD] Devices with the High-Speed Analog Comparator with the Slope Compensation DAC peripheral. See the device's header file to determine if the function is available.

**Requires:**
---

**Examples:**

```
[PCD]  setup_comparator_dac(COMP_COMMON_DAC_ENABLE |
    COMP_DAC_CLK_SRC_FPLLO, 85, 138);
```

**See Also:**
Analog Comparator Overview, setup_comparator(), setup_comparator_slope(), dac_write()

## setup_comparator_filter( )

**Syntax:**
[PCD] setup_comparator (**comparator, *mode***);

**Parameters:**
[PCD] **comparator** - constant specifying which comparator filter to setup.

[PCD] ***mode* -** constants specifying the settings to setup the specified comparator's filter. See the device's .h file for all options. Some typical options include:

COMP_FILTER_ENABLE
COMP_FILTER_CLK_T3
COMP_FILTER_CLK_T2
COMP_FILTER_CLK_FOSC
COMP_FILTER_CLK_INTERNAL
COMP_FILTER_CLK_DIV_BY_4
COMP_FILTER_CLK_DIV_BY_2
COMP_FILTER_CLK_DIV_BY_1

**Returns:**
---

**Function:**
[PCD] Configures the voltage comparator's digital filter.

**Availability:**
[PCD] Devices with a comparator module that has a digital filter. See the device's header file to determine if the device has a digital filter as part of the comparator module.

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
[PCD]  setup_comparator_filter(1,COMP_FILTER_ENABLE|
    COMP_FILTER_CLK_FOSC|COMP_FILTER_CLK_DIV_BY_4);
```

**See Also:**
Analog Comparator Overview, setup_comparator(), setup_comparator_mask()

## setup_comparator_mask( )

**Syntax:**
[PCD] setup_comparator_mask (**comparator, *mode, [input1], [input2], [input3]***);

**Parameters:**
[PCD] **comparator** - constant specifying which comparator filter to setup.

[PCD] ***mode* -** constants specifying the settings to setup the specified comparator's mask registers. See the device's .h file for all options. Some typical options include:

COMP_MASK_COMP_HIGH
COMP_MASK_COMP_LOW
COMP_MASK_MAI_CONNECTED_TO_OR
COMP_MASK_INVERTED_MAI_CONNECTED_TO_OR
COMP_MASK_MAI_CONNECTED_TO_AND
COMP_MASK_INVERTED_MAI_CONNECTED_TO_AND

[PCD] ***input1, input2, input3*** - optional parameters specifying the inputs to mask. See the device's .h file for all options. Some typical options include:

COMP_MASK_INPUT_PWM3H
COMP_MASK_INPUT_PWM3L
COMP_MASK_INPUT_PWM2H
COMP_MASK_INPUT_PWM2L
COMP_MASK_INPUT_PWM1H
COMP_MASK_INPUT_PWM1L

**Returns:**
---

**Function:**
[PCD] Configures the voltage comparator's output blanking function.

**Availability:**
*[PCD]* Devices with a comparator module that has a output blanking function. See the device's header file to determine if the device has an output blanking function as part of the comparator module.

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
[PCD]  setup_comparator_mask(1,COMP_MASK_COMP_LOW|
    COMP_MASK_MAI_CONNECTED_TO_AND, COMP_MASK_INPUT_PWM1H;
```

**See Also:**
Analog Comparator Overview, setup_comparator(), setup_comparator_filter()

## *[PCD]* setup_comparator_slope( )

**Syntax:**
*[PCD]* setup_comparator_slope(**comparator, settings, rate**);

**Parameters:**
*PCD]* **comparator** - constant specifying which comparator to setup.

*[PCD]* **settings** - constants specifying the settings to setup the comparator and DAC. See the device's header file for defines that can be used with this parameter.

*[PCD]* ***rate -*** 16-bit value used to set slope ramp rate, the value is in 12.4 format.

**Returns:**
---

**Function:**
*[PCD]* Used to setup the Analog Comparator with Slope Compensation DAC peripheral slope settings.

**Availability:**
*[PCD]* Devices with the High-Speed Analog Comparator with the Slope Compensation DAC peripheral. See the device's header file to determine if the function is available.

**Requires:**
---

**Examples:**

```
[PCD] setup_comparator_slope(1,COMP_ENABLE_SLOPE_FUNCTION |
    COMP_SLOPE_START_SIG_PWM1_TRIG_1 |
COMP_SLOPE_STOP_B_SIG_COMP1 |
   COMP_SLOPE_STOP_A_SIG_PWM1_TRIG_1, 500000);
```

**See Also:**
Analog Comparator Overview, setup_comparator(), setup_comparator_dac(), dac_write()

## setup_comparator_x( )

**Syntax:**
setup_comparator_1(***mode***);
setup_comparator_2(***mode***);
setup_comparator_3(***mode***);
setup_comparator_4(***mode***);
setup_comparator_5(***mode***);
setup_comparator_6(***mode***);
setup_comparator_7(***mode***);
setup_comparator_8(***mode***);

**Parameters:**
***mode* -** to setup the comparator in. Valid options are device dependent. See the device's header file for all valid options.

**Returns:**
---

**Function:**
Used to setup one of the Analog Comparator modules.

**Availability:**
On most devices that have more than three Analog Comparator modules.

**Requires:**
---

**Examples:**

```
setup_comparator_1(CP1_A1_A0 | CP1_INVERT);
```

**See Also:**
setup_comparator( ), Analog Comparator

## setup_compare( )

**Syntax:**
setup_compare(***x, mode***)

**Parameters:**
***mode -*** is defined by the constants in the devices .h file

***x -*** is 1-16 and specifies which OC pin to use.

**Returns:**
---

**Function:**
This function specifies how the output compare module is going to function based on the value of ***mode***. The device specific options are listed in the device .h file.

**Availability:**
Available only on devices with Output Compare Modules

**Requires:**
---

**Examples:**

```
                                                    // Pin OC1 will be set
  when timer 2
                                                    // is equal to 0xF000
  setup_timer2(TMR_INTERNAL | TIMER_DIV_BY_16);
  set_compare_time(1, 0xF000);
  setup_compare(1, COMPARE_SET_ON_MATCH | COMPARE_TIMER2);
```

**See Also:**
set_compare_time(), set_pwm_duty(), setup_capture(), Output Compare / PWM Overview

## setup_counters( )

**Syntax:**
setup_counters (***rtcc_state***, ***ps_state***)

**Parameters:**

***rtcc_state -*** may be one of the constants defined in the devices .h file.

RTCC_INTERNAL
RTCC_EXT_L_TO_H
RTCC_EXT_H_TO_L

***ps_state -*** may be one of the constants defined in the devices .h file.

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:**

---

**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);
```

**See Also:**
setup wdt(), setup_timer 0(), see header file for device selected

## setup_crc(mode)

**Syntax:**
setup_crc(***polynomial terms***)

**Parameters:**
***polynomial*** - This will setup the actual polynomial in the CRC engine. The power of each term is passed separated by a comma. 0 is allowed, but ignored. The following define is added to the device's header file (32-bit CRC Moduel Only), to enable little-endian shift direction:

CRC_LITTLE_ENDIAN

**Returns:**
---

**Function:**
Configures the CRC engine register with the polynomial.

**Availability:**
Devices with built in CRC module

**Requires:**
---

**Examples:**

```
setup_crc (12, 5);                    // CRC Polynomial is X^12 + X^5 + 1

setup_crc(16, 15, 3, 1);             // CRC Polynomial is X^16 + X^15 + X^3 +
X^1+ 1
```

**Example Files:**
ex.c

**See Also:**
crc_init(); crc_calc(); crc_calc8()

## setup_cog( )setup_cog2( )

**setup_cog3( )**
**setup_cog4( )**

**Syntax:**
setup_cog(**mode**, [**shutdown]**);
setup_cog(mode, [shutdown], [sterring]);

**Parameters:**
**mode**- the setup of the COG module. See the device's .h file for all options. Some typical options include:

COG_ENABLED
COG_DISABLED
COG_CLOCK_HFINTOSC
COG_CLOCK_FOSC

**shutdown**- the setup for the auto-shutdown feature of COG module. See the device's .h file for all the options. Some typical options include:

COG_AUTO_RESTART
COG_SHUTDOWN_ON_C1OUT
COG_SHUTDOWN_ON_C2OUT

**steering**- optional parameter for steering the PWM signal to COG output pins and/or selecting the COG pins static level. Used when COG is set for steered PWM or synchronous steered PWM modes. Not available on all devices, see the device's .h file if available and for all options. Some typical options include:

COG_PULSE_STEERING_A
COG_PULSE_STEERING_B
COG_PULSE_STEERING_C
COG_PULSE_STEERING_D

**Returns:**
---

**Function:**
Sets up the Complementary Output Generator (COG) module, the auto-shutdown feature of the module and if available steers the signal to the different output pins.

**Availability:**
Devices with built in COG module

**Requires:**
---

**Examples:**

```
setup_cog(COG_ENABLED | COG_PWM | COG_FALLING_SOURCE_PWM3 |
COG_RISING_SOURCE_PWM3, COG_NO_AUTO_SHUTDOWN,
COG_PULSE_STEERING_A | COG_PULSE_STEERING_B);
```

**See Also:**
set_cog_dead_band(), set_cog_phase(), set_cog_blanking(), cog_status(), cog_restart()

## setup_cwg( ) setup_cwg2( ) setup_cwg3( )

**Syntax:**
setup_cwg(**mode**,**shutdown**,**dead_time_rising**,**dead_time_falling**)

**Parameters:**
***mode*** - the setup of the CWG module. See the device's .h file for all options. Some typical options include:

CWG_ENABLED
CWG_DISABLED
CWG_OUTPUT_B
CWG_OUTPUT_A

***shutdown*** - the setup for the auto-shutdown feature of CWG module. See the device's .h file for all the options. Some typical options include:

CWG_AUTO_RESTART
CWG_SHUTDOWN_ON)COMP1
CWG_SHUTDOWN_ON_FLT
CWG_SHUTDOWN_ON_CLC2

***dead_time_rising*** - value specifying the dead time between A and B on the rising edge. (0-63)

***dead_time_rising*** - value specifying the dead time between A and B on the falling edge. (0-63)

**Returns:**
---

**Function:**
Sets up the CWG module, the auto-shutdown feature of module and the rising and falling dead times of the module.

**Availability:**
Devices with built in CWG module

**Requires:**
---

**Examples:**

```
setup_cwg(CWG_ENABLED|CWG_OUTPUT_A|CWG_OUTPUT_B|CWG_INPUT_PWM1,CWG_SHUTDOWN_ON_F
LT,60,30);
```

**See Also:**
cwg_status( ), cwg_restart( )

## [PCD] setup_current_source()

**Syntax:**
setup_current_source(**mode**);

**Parameters:**
**mode** - setup the Constant Current Source module. Valid options are device dependent. See the device's header file for all options.

**Returns:**
---

**Function:**
Used to setup the Constant Current Source module.

**Availability:**
Devices that have a Constant Current Source module.

**Requires:**
------

**Examples:**

```
setup_current_source(CURRENT_SOURCE_ENABLED | CURRENT_SOURCE_D5);
```

## setup_dac( )

**Syntax:**
setup_dac(**mode**);
setup_dac2(**mode**);
setup_dac3(**mode**);

setup_dac4(**mode**);
setup_dac5(**mode**);
setup_dac6(**mode**);
setup_dac7(**mode**);
setup_dac8(**mode**);
*[PCD]* setup_dac(**mode, divisor**);
*[PCD]* setup_dac(**module, mode**);

**Parameters:**
***mode -*** The mode to setup the DAC module in. The valid options vary depending on the device. See the device's header file for all options.

*[PCD]* ***divisor -*** Divides the provided clock.

*[PCD]* ***module -*** DAC module setup.

**Returns:**
---

**Function:**
Setup the DAC module.

**Availability:**
Devices with a digital-to-analog converter (DAC).

**Requires:**
---

**Examples:**
```
setup_dac(DAC_VSS_VDD | DAC_OUTPUT);

[PCD]  setup_dac(DAC_RIGHT_ON, 5);
     setup_dac(1, DAC_ON)
```

**See Also:**
dac_write, DAC. See header file for selected device.

## setup_dci( )

**Syntax:**
setup_dci***(configuration, data size, rx config, tx config, sample rate)***;

**Parameters:**
***configuration*** - Specifies the configuration the Data Converter Interface should be initialized into, including the mode of transmission and bus properties. The following constants may be combined ( OR'd) for this parameter:

CODEC_MULTICHANNEL
CODEC_I2S· CODEC_AC16
CODEC_AC20· JUSTIFY_DATA· DCI_MASTER
DCI_SLAVE· TRISTATE_BUS· MULTI_DEVICE_BUS
SAMPLE_FALLING_EDGE· SAMPLE_RISING_EDGE
DCI_CLOCK_INPUT· DCI_CLOCK_OUTPUT

***data size*** – Specifies the size of frames and words in the transmission:

DCI_xBIT_WORD: x may be 4 through 16
DCI_xWORD_FRAME: x may be 1 through 16
DCI_xWORD_INTERRUPT: x may be 1 through 4

***rx config***- Specifies which words of a given frame the DCI module will receive (commonly used for a multi-channel, shared bus situation)

RECEIVE_SLOTx: x May be 0 through 15
RECEIVE_ALL· RECEIVE_NONE

***tx config***- Specifies which words of a given frame the DCI module will transmit on.

TRANSMIT_SLOTx: x May be 0 through 15
TRANSMIT _ALL
TRANSMIT _NONE

***sample rate*** - The desired number of frames per second that the DCI module should produce. Use a numeric value for this parameter. Keep in mind that not all rates are achievable with a given clock. Consult the device datasheet for more information on selecting an adequate clock.

**Returns:**
---

**Function:**
Configures the DCI module.

**Availability:**
Only available on devices with DCI peripheral.

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
dci_initialize((I2S_MODE|DCI_MASTER|DCI_CLOCK_OUTPUT|SAMPLE_RISING_EDGE|UNDERFLO
W_LAST|
                MULTI_DEVICE_BUS,DCI_1WORD_FRAME|DCI_16BIT_WORD|DCI_2WORD_INTERR
UPT,
               RECEIVE_SLOT0|RECEIVE_SLOT1, TRANSMIT_SLOT0|TRANSMIT_SLOT1, 44100
);
```

**See Also:**
DCI Overview, dci start( ), dci write( ), dci read( ), dci transmit ready( ), dci data received()

## setup_dedicated_adc( )

**Syntax:**
setup_dedicated_adc(**core, mode**);

**Parameters:**
**core** - the dedicated ADC core to setup

**mode** - the mode to setup the dedicated ADC core in. See the device's .h file all options. Some typical options include:

ADC_DEDICATED_CLOCK_DIV_2
ADC_DEDICATED_CLOCK_DIV_6
ADC_DEDICATED_TAD_MUL_2
ADC_DEDICATED_TAD_MUL_3

**Returns:**
---

**Function:**
Configures one of the dedicated ADC core's clock speed and sample time.
Function should be called after the setup_adc() function.

**Availability:**
Only available on dsPIC33EPxxGSxxx family of devices.

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
setup_dedicated_adc(0,ADC_DEDICATED_CLOCK_DIV_2|ADC_DEDICATED_TA
D_MUL_1025)
```

**See Also:**
setup_adc(), setup_adc_ports(), set_adc_channel(), read_adc(), adc_done(), set_dedicated_adc_channel(), ADC Overview

## setup_dma( )

**Syntax:**
setup_dma(**channel, start_trigger, abort_trigger)**;
*[PCD]* setup_dma**(channel, peripheral,mode)**;
*[PCD]* setup_dma(**channel, trigger, mode)**;

**Parameters:**
***channel*** -The DMA channel to setup.

***start_trigger*** - The trigger source to cause the DMA channel to start the transfer when HW trigger is enabled. See header file for all possible sources.

***abort_trigger*** - The trigger source to cause the DMA channel to abort the transfer when HW abort trigger is enabled. See header file for all possible sources.

*[PCD]* ***peripheral*** - The peripheral that the DMA channel transfers data to and from. Constants for setting the trigger source are defined in the device's .h file, see header file for all possible peripherals.

*[PCD]* ***trigger*** - The trigger source to cause the DMA channel to start the transfer. Constants for setting the trigger source are defined in the device's header file, see header file for all possible sources.

*[PCD]* ***mode*** - The mode to use for the DMA transfers. Constants for setting the mode are defined in the device's header file, see header file for all possible options.

**Returns:**
---

**Function:**
Configures the DMA peripheral to copy data from one location to another.

**Availability:**
Devices that have a DMA peripheral. *[PCD]* The version of the function depends on the type of DMA peripheral it has. Use ***getenv("DMA")*** to determine the type the device has. It will return 0 for no DMA peripheral, 1 for Type 1 and 2 for Type 2. For devices with

Type 1 uses first version of the function and for devices with Type 2 uses second version of the function.

**Requires:**
---

**Examples:**

```
[PCD]
setup_dma(0,DMA_IN_UART1,DMA_BYTE);     // Type 1
setup_dma(0,DMA_TRIGGER_RDA,DMA_BYTE|
DMA_RELOAD_ADDRESS);                    // Type 2

setup_dma(1, DMA_TRIGGER_RDA, DMA_TRIGGER_NONE);
```

**Example Files:**
ex_dma_uart_rx.c

**See Also:**
dma_start(), dma_status()

## setup_dmt( )

**Syntax:**
setup_dmt***(mode, max_time, window_time)***;

**Parameters:**
***mode*** - This sets if whether the DMT is always enabled or can be enabled and disabled in software. The following defines are made in the device's header file for setting the following:

DMT_SOFTWARE // DMT can be enabled and disabled in software.
DMT_ENABLED // DMT is ways enabled.

***max_time*** - A 32-bit constant value for setting the count that causes a DMT event to occur.

***window_time*** - A 32-bit constant value for setting the count value at which it is possible to clear the DMT count value.

**Returns:**
---

**Function:**
Sets up the Deadman Timer (DMT) peripheral. This function does not generate any assembly code, it only causes the DMT configuration fuses to be set to the specified values.

**Availability:**
Only on devices that have the DMT peripheral.

**Requires:**
---

**Examples:**
```
  //Setup DMT peripheral to be enabled and disabled in software
  //with a max count of 50000 and can be cleared after count
  //reaches 10000.
  setup_dmt(DMT_SOFTWARE, 50000, 10000);
```

**See Also:**
clear_dmt(), read_dmt(), disable_dmt(), enable_dmt(), dmt_status()

## setup_dsm()

**Syntax:**
setup_dsm(**enable**);
setup_dsm2(**enable**);
setup_dsm3(**enable**);
setup_dsm4(**enable**);
setup_dsm(**mode**, **source**, **carrier**);
setup_dsm2(**mode**, **source**, **carrier**);
setup_dsm3(**mode**, **source**, **carrier**);
setup_dsm4(**mode**, **source**, **carrier**);

**Parameters:**
**enable** – a 1-bit constant used to enable and disable the DSM module. If 1 is passed as the parameter, the DSMx module is enabled, and if 0 is passed as the parameter, the DSM module is disabled.

**mode** - the mode to setup the DSM module. Valid options vary by device. See the device's header file for all options.

**source** - used to set the signal source for the DSM module. Valid options vary by device. See the device's header file for all options.

**carrier** - used to set the high and low level carriers for the DSM module. Valid options vary by device. See the device's header file for all options.

**Returns:**
---

**Function:**
Used to setup the DSM module.

**Availability:**
Devices that have a Data Signal Modulator (DSM) module

**Requires:**
------

**Examples:**

```
setup_dsm(DSM_ENABLED, DSM_SOURCE_U1TX, DSM_CARRIER_LOW_CCP1 |
DSM_CARRIER_HIGH_CCP2);
```

**See Also:**
Data Signal Modulator Overview

## setup_external_memory( )

**Syntax:**
setup_external_memory**(mode)**;

**Parameters:**
***mode* -** is one or more constants from the device header file OR'ed together.

**Returns:**
---

**Function:**
Sets the mode of the external memory bus.

**Availability:**
Devices that allow external memory bus.

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
setup_external_memory(EXTMEM_WORD_WRITE|EXTMEM_WAIT_0 );
setup_external_memory(EXTMEM_DISABLE);
```

**See Also:**
WRITE PROGRAM EEPROM() , WRITE PROGRAM MEMORY(), External Memory Overview

## setup_high_speed_adc( )

**Syntax:**
setup_external_memory**(mode)**;

**Parameters:**
***mode* -** Analog to digital mode. The valid options vary depending on the device. See the devices .h file for all options. Some typical options include:

ADC_OFF
ADC_CLOCK_DIV_1
ADC_HALT_IDLE (The ADC will not run when device is idle)

**Returns:**
---

**Function:**
Configures the High-Speed ADC clock speed and other High-Speed ADC options including, when the ADC interrupts occurs, the output result format, the conversion order, whether the ADC pair is sampled sequentially or simultaneously, and whether the dedicated sample and hold is continuously sampled or samples when a trigger event occurs.

**Availability:**
dsPIC33FJxxGSxxx devices

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
setup_high_speed_adc_pair(0, INDIVIDUAL_SOFTWARE_TRIGGER);
setup_high_speed_adc(ADC_CLOCK_DIV_4);
read_high_speed_adc(0, START_AND_READ, result);
setup_high_speed_adc(ADC_OFF);
```

**See Also:**
setup_high_speed_adc_pair(), read_high_speed_adc(), high_speed_adc_done()

## setup_high_speed_adc_pair( )

**Syntax:**
setup_high_speed_adc_pair(***pair, mode***);

**Parameters:**
***pair*** – The High-Speed ADC pair number to setup, valid values are 0 to total number of ADC pairs. 0 sets up ADC pair AN0 and AN1, 1 sets up ADC pair AN2 and AN3, etc.

***mode*** – ADC pair mode. The valid options vary depending on the device. See the devices .h file for all options. Some typical options include:

INDIVIDUAL_SOFTWARE_TRIGGER
GLOBAL_SOFTWARE_TRIGGER
PWM_PRIMARY_SE_TRIGGER
PWM_GEN1_PRIMARY_TRIGGER
PWM_GEN2_PRIMARY_TRIGGER

**Returns:**
---

**Function:**
Sets up the analog pins and trigger source for the specified ADC pair. Also sets up whether ADC conversion for the specified pair triggers the common ADC interrupt.

If zero is passed for the second parameter the corresponding analog pins will be set to digital pins.

**Availability:**
dsPIC33FJxxGSxxx devices

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
setup_high_speed_adc_pair(0,INDIVIDUAL_SOFTWARE_TRIGGER);
setup_high_speed_adc_pair(1,GLOBAL_SOFTWARE_TRIGGER);
setup_high_speed_adc_pair(2,0)                //sets AN4 and AN5
as digital pins
```

**See Also:**
setup_high_speed_adc(), read_high_speed_adc(), high_speed_adc_done()

## setup_hspwm( ) setup_hspwm_secondary( )

**Syntax:**
setup_hspwm(**mode, value**);
setup_hspwm_secondary(**mode, value**); //if available

**Parameters:**
***mode*** - Mode to setup the High Speed PWM module in. The valid options vary depending on the device. See the device's .h file for all options. Some typical options include:

HSPWM_ENABLED
HSPWM_HALT_WHEN_IDLE
HSPWM_CLOCK_DIV_1

***value*** - 16-bit constant or variable to specify the time bases period.

**Returns:**
---

**Function:**
Enable the High Speed PWM module and set up the Primary and Secondary Time base of the module.

**Availability:**
Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**
Constants are defined in the device's .h file

**Examples:**
```
setup_hspwm(HSPWM_ENABLED | HSPWM_CLOCK_DIV_BY4, 0x8000);
```

**See Also:**
setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_blanking(), setup_hspwm_trigger(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock(), setup_hspwm_secondary()

## setup_hspwm_blanking( )

**Syntax:**
setup_hspwm_blanking(**unit, settings, delay**);

**Parameters:**
***unit*** - The High Speed PWM unit to set.

***settings*** - Settings to setup the High Speed PWM Leading-Edge Blanking. The valid options vary depending on the device. See the device's header file for all options. Some typical options include:

HSPWM_RE_PWMH_TRIGGERS_LE_BLANKING
HSPWM_FE_PWMH_TRIGGERS_LE_BLANKING
HSPWM_RE_PWML_TRIGGERS_LE_BLANKING
HSPWM_FE_PWML_TRIGGERS_LE_BLANKING
HSPWM_LE_BLANKING_APPLIED_TO_FAULT_INPUT
HSPWM_LE_BLANKING_APPLIED_TO_CURRENT_LIMIT_INPUT

***delay*** - 16-bit constant or variable to specify the leading-edge blanking time.

**Returns:**
---

**Function:**
Sets up the analog pins and trigger source for the specified ADC pair. Also sets up whether ADC conversion for the specified pair triggers the common ADC interrupt.

If zero is passed for the second parameter the corresponding analog pins will be set to digital pins.

**Availability:**
Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**
---

**Examples:**
```
setup_hspwm_blanking(HSPWM_RE_PWMH_TRIGGERS_LE_BLANKING, 10);
```

**See Also:**
setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_blanking(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock() setup_hspwm(), setup_hspwm_secondary(), setup_high_speed_adc(), read_high_speed_adc(), high_speed_adc_done()

## setup_hspwm_chop_clock( )

**Syntax:**
setup_hspwm_chop_clock(**settings**);

**Parameters:**
***unit*** - The High Speed PWM unit to set.

***settings*** - a value from 1 to 1024 to set the chop clock divider. Also one of the following can be or'd with the value:
HSPWM_CHOP_CLK_GENERATOR_ENABLED
HSPWM_CHOP_CLK_GENERATOR_DISABLED

**Returns:**
---

**Function:**
Setup and High Speed PWM Chop Clock Generator and divisor.

**Availability:**
Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**
---

**Examples:**
```
setup_hspwm_chop_clock(HSPWM_CHOP_CLK_GENERATOR_ENABLED|32);
```

**See Also:**
setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_blanking(), setup_hspwm_trigger(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_unit_chop_clock(), setup_hspwm(), setup_hspwm_secondary()

## setup_hspwm_current_limit( )

**Syntax:**
setup_hspwm_current_limit(**unit, settings**);

**Parameters:**
***unit*** - The High-Speed PWM unit to setup.

***settings*** - An int32 value to setup the PWM Generator Current Limit PCI registers. See the device's header file for valid defines that can be used with function.

**Returns:**
---

**Function:**
To setup the High-Speed PWM (HSPWM) Current Limit PCI registers.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
setup_hswm_current_limit(1, HSPWM_PCI_SRC_MASTER_PCI10 |
HSPWM_ACCEPTANCE_QUALIFIER_PWM_TRIGGERED |
HSPWM_TERMINATION_QUALIFIER_DUTY_CYCLE |
HSPWM_PCI_ACCEPTANCE_LATCHED_ANY_EDGE);
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## setup_hspwm_event( )

**Syntax:**
setup_hspwm_event(**unit, settings_1, settings_h**);

**Parameters:**
***unit*** - The High-Speed PWM unit to setup.

***settings_1*** - An int16 constant value to setup the PWM Generator Event low register settings. See the device's header file for valid defines that can be used with function.

***settings_h*** - An int16 constant value to setup the PWM Generator Event high register settings. See the device's header file for valid defines that can be used with function.

**Returns:**
---

**Function:**
To setup the High-Speed PWM (HSPWM) Event registers.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
setup_hswm_EVENT(3, HSPWM_EVENT_TRIGGER_EOC |
HSPWM_EVENT_UPDATE_WRITE_TO_DUTY_CYCLE |
HSPWM_EVENT_ADC_TRIGGER_1_PGxTRIGB_SRC_ENABLED | 30,
HSPWM_EVENT_INTERRUPT_DISABLED |
HSPWM_EVENT_FEED_FORWARD_INTERRUPT_ENABLED | 10);
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_fault(), setup_hspwm_current_limit(),setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## setup_hspwm_fault( )

**Syntax:**
setup_hspwm_fault(**unit, settings**);

**Parameters:**
***unit*** - The High-Speed PWM unit to setup.

***settings*** - An int32 value to setup the PWM Generator Fault PCI registers. See the device's header file for valid defines that can be used with function.

**Returns:**
---

**Function:**
To setup the High-Speed PWM (HSPWM) Fault PCI registers.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
setup_hswm_fault(1, HSPWM_PCI_SRC_MASTER_PCI10 |
HSPWM_ACCEPTANCE_QUALIFIER_PWM_TRIGGERED |
HSPWM_TERMINATION_QUALIFIER_DUTY_CYCLE |
HSPWM_PCI_ACCEPTANCE_LATCHED_ANY_EDGE);
```

**See Also:**
setup_hspwm(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## setup_hspwm_feed_forward( )

**Syntax:**
setup_hspwm_feed_forward(**unit, settings**);

**Parameters:**
***unit*** - The High-Speed PWM unit to setup.

***settings*** - An int32 value to setup the PWM Generator Feed Forward PCI registers. See the device's header file for valid defines that can be used with function.

**Returns:**
---

**Function:**
To setup the High-Speed PWM (HSPWM) Feed Forward PCI registers.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
setup_hswm_feed_forward(1, HSPWM_PCI_SRC_MASTER_PCI10 |
HSPWM_ACCEPTANCE_QUALIFIER_PWM_TRIGGERED |
HSPWM_TERMINATION_QUALIFIER_DUTY_CYCLE |
HSPWM_PCI_ACCEPTANCE_LATCHED_ANY_EDGE);
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## setup_hspwm_logic_x( )

**Syntax:**
setup_hspwm_logic_a(**settings**);
setup_hspwm_logic_b(**settings**);

setup_hspwm_logic_c(**settings**);
setup_hspwm_logic_d(**settings**);
setup_hspwm_logic_e(**settings**);
setup_hspwm_logic_f(**settings**);

**Parameters:**
***settings*** - An int16 value to setup the PWM Logic control to. See the device's header file for valid defines that can be used with function.

**Returns:**
---

**Function:**
To setup the High-Speed PWM (HSPWM) Logic control registers.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
setup_hswm_logic_a(HSPWM_LOGIC_SRC1_PWM1H |
HSPWM_LOGIC_SRC2_PWM4H | HSPWM_LOGIC_SRC1_OR_SRC2 |
HSPWM_LOGIC_ASSIGNED_TO_PWM2);
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(),setup_hspwm_feed_forward(), setup_hspwn_sync(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update()

## setup_hspwm_sync( )

**Syntax:**
setup_hspwm_sync(**unit, settings**);

**Parameters:**
***unit*** - The High-Speed PWM unit to setup.

***settings*** - An int32 value to setup the PWM Generator Sync PCI registers. See the device's header file for valid defines that can be used with function.

**Returns:**
---

**Function:**
To setup the High-Speed PWM (HSPWM) Sync PCI registers.

**Availability:**
On devices that have the HSPWM peripheral with Fine Edge Placement. See the device's header file to determine if functions are available.

**Requires:**
---

**Examples:**
```
setup_hswm_sync(1, HSPWM_PCI_SRC_MASTER_PCI10 |
HSPWM_ACCEPTANCE_QUALIFIER_PWM_RIGGERED |
HSPWM_TERMINATION_QUALIFIER_DUTY_CYCLE |
HSPWM_PCI_ACCEPTANCE_LATCHED_ANY_EDGE);
```

**See Also:**
setup_hspwm(), setup_hspwm_event_output_x(), setup_hspwm_logic_x(), setup_hspwm_unit(), setup_hspwm_blanking(), setup_hspwm_event(), setup_hspwm_fault(), setup_hspwm_current_limit(), setup_hspwm_feed_forward(), set_hspwm_scaling(), set_hspwm_overrride(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_period(), set_hspwm_duty_adjustment(), set_hspwm_trigger_x(), get_hspwm_feedback(), get_hspwm_capture(), get_hspwm_status(), hspwm_trigger_pwm(), hspwm_stop_pwm(), hspwm_do_capture(), hspwm_update(),

## setup_hspwm_trigger( )

**Syntax:**
setup_hspwm_trigger(**unit, [start_ delay], [divider], [trigger_value], [strigger_value]**);

**Parameters:**
***unit*** - The High Speed PWM unit to set.

***start_delay*** - Optional value from 0 to 63 specifying then umber of PWM cycles to wait before generating the first trigger event. For some devices, one of the following may be optional or'd in with the value:

HSPWM_COMBINE_PRIMARY_AND_SECONDARY_TRIGGER
HSPWM_SEPERATE_PRIMARY_AND_SECONDARY_TRIGGER

***divider*** - optional value from 1 to 16 specifying the trigger event divisor.

***trigger_value*** - optional 16-bit value specifying the primary trigger compare time.

***strigger_value*** - optional 16-bit value specifying the secondary trigger compare time.

**Returns:**
---

**Function:**
Sets up the High Speed PWM Trigger event.

**Availability:**
Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**
---

**Examples:**
```
setup_hspwm_trigger(1, 10, 1, 0x2000);
```

**See Also:**
setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_trigger(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock(), setup_hspwm(), setup_hspwm_secondary()

## setup_hspwm_unit( )

**Syntax:**
setup_hspwm_unit(**unit, mode, [dead_time], [alt_dead_time]**);
set_hspwm_duty(**unit, primary, [secondary]**);

**Parameters:**
***unit*** - The High Speed PWM unit to set.

***mode*** - Mode to setup the High Speed PWM unit in. The valid option vary depending on the device. See the device's header file for all options. Some typical options include:
HSPWM_ENABLE

HSPWM_ENABLE_H
HSPWM_ENABLE_L
HSPWM_COMPLEMENTARY
HSPWM_PUSH_PULL

***dead_time*** - Optional 16-bit constant or variable to specify the dead time for this PWM unit, defaults to 0 if not specified.

***alt_dead_time*** - Optional 16-bit constant or variable to specify the alternate dead time for this PWM unit, default to 0 if not specified.

**Returns:**
---

**Function:**
Sets up the specified High Speed PWM unit.

**Availability:**
Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**
Constants are defined in the device's .h file

**Examples:**
```
setup_hspwm_unit(1,HSPWM_ENABLE|SHPWM_COMPLEMENTARY, 100,100);
```

**See Also:**
set_hspwm_phase(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_blanking(), setup_hspwm_trigger(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm_unit_chop_clock(), setup_hspwm(), setup_hspwm_secondary()

## setup_hspwm_unit_chop_clock( )

**Syntax:**
setup_hspwm_unit_chop_clock(**unit, settings**);

**Parameters:**
***unit*** - the High Speed PWM unit chop clock to setup.

***settings*** - a settings to setup the High Speed PWM unit chop clock. The valid options vary depending on the device. See the device's .h file for all options. Some typical options include:

HSPWM_PWMH_CHOPPING_ENABLED
HSPWM_PWML_CHOPPING_ENABLED
HSPWM_CHOPPING_DISABLED
HSPWM_CLOP_CLK_SOURCE_PWM2H
HSPWM_CLOP_CLK_SOURCE_PWM1H
HSPWM_CHOP_CLK_SOURCE_CHOP_CLK_GENERATOR

**Returns:**
---

**Function:**
Setup and High Speed PWM unit's Chop Clock

**Availability:**
Only on devices with a built-in High Speed PWM module (dsPIC33FJxxGSxxx, dsPIC33EPxxxMUxxx, dsPIC33EPxxxMCxxx, and dsPIC33EVxxxGMxxx devices)

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
setup_hspwm_unit_chop_clock(1,HSPWM_PWMH_CHOPPING_ENABLED|
                            HSPWM_PWML_CHOPPIJNG_ENABLED|
                            HSPWM_CLOP_CLK_SOURCE_PWM2H);
```

**See Also:**
setup_hspwm_unit(), set_hspwm_phase(), set_hspwm_duty(), set_hspwm_event(), setup_hspwm_blanking(), setup_hspwm_trigger(), set_hspwm_override(), get_hspwm_capture(), setup_hspwm_chop_clock(), setup_hspwm(), setup_hspwm_secondary()

## setup_lcd( )

**Syntax:**
setup_lcd (***mode***, ***prescale***, [***segments0_31***]**,[segments32_47])**;

**Parameters:**
***mode -*** may be any of the following constants to enable the LCD and may be or'ed with other constants in the devices *.*h* file:

LCD_DISABLED, LCD_STATIC, LCD_MUX12, LCD_MUX13, LCD_MUX14

***prescale -*** may be 1-16 for the LCD clock**.**

***segments0-31 -*** may be any of the following constants or'ed together when using the PIC16C92X series of chips::

SEG0_4, SEG5_8, SEG9_11, SEG12_15, SEG16_19, SEG20_26, SEG27_28, SEG29_31
ALL_LCD_PINS

When using the PIC16F/LF1xxx or PIC18F/LFxxxx series of chips, each of the segments are enabled individually. A value of 1 will enable the segment, 0 will disable it and use the pin for normal I/O operation.

***segments 32-47 -*** when using a chip with more than 32 segments, this enables segments 32-47. A value 1 will enable the segment, 0 will disable it. Bit 0 corresponds to segment 32 and bit 15 corresponds to segment 47.

**Returns:**

---

**Function:**
Initialize the LCD Driver Module on the PIC16C92X and PIC16F/LF193X series of devices.

**Availability:**
Only on devices with built-in LCD Driver Module hardware.

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
setup_lcd(LCD_MUX14|LCD_STOP_ON_SLEEP,2,ALL_LCD_PINS);
                                                  // PIC16C92X
setup_lcd(LCD_MUX13|LCD_REF_ENABLED|LCD_B_HIGH_POWER,0,0xFF0429)
;
                                                  // PIC16F/LF193X –
                                                  //Enables Segments
                                                  //0,3,5,10,16,17,18,
19,20,21,22,23
```

**Example Files:**
ex_92lcd.c

**See Also:**
lcd_symbol(), lcd_load(), lcd_contrast( ), Internal LCD Overview

## setup_low_volt_detect( )

**Syntax:**
setup_low_volt_detect(**mode**)

**Parameters:**
**mode** may be one of the constants defined in the devices .h file.

LVD_LVDIN
LVD_45
LVD_42
LVD_40
LVD_38
LVD_36
LVD_35
LVD_33
LVD_30
LVD_28
LVD_27
LVD_25
LVD_23
LVD_21
LVD_19

One of the following may be or'ed(via |) with the above if high voltage detect is also available in the device

LVD_TRIGGER_BELOW
LVD_TRIGGER_ABOVE

**Returns:**
---

**Function:**
This function controls the high/low voltage detect module in the device. The mode constants specifies the voltage trip point and a direction of change from that point (available only if high voltage detect module is included in the device). If the device experiences a change past the trip point in the specified direction the interrupt flag is set and if the interrupt is enabled the execution branches to the interrupt service routine.

**Availability:**
Only available with devices that have the high/low voltage detect module.

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
setup_low_volt_detect( LVD_TRIGGER_BELOW | LVD_36 );  //This
would trigger the
                                                      //interrupt
when the voltage
                                                      //is below
3.6 volts
```

## setup_motor_pwm( )

**Syntax:**
setup_motor_pwm(***pwm***,***options***, ***timebase***);
setup_motor_pwm(***pwm***,***options***,***prescale***,***postscale***,***timebase***)

**Parameters:**
***pwm*** - Defines the pwm module used.

***Options*** - The mode of the power PWM module. See the devices .h file for all options

***timebase*** - This parameter sets up the PWM time base pre-scale and post-scale.

***prescale*** - This will select the PWM timebase prescale setting

***postscale*** - This will select the PWM timebase postscale setting

**Returns:**
---

**Function:**
Configures the motor control PWM module.

**Availability:**
Devices that have the motor control PWM unit.

**Requires:**
---

**Examples:**

```
setup_motor_pwm(1,MPWM_FREE_RUN | MPWM_SYNC_OVERRIDES,
timebase);
```

**See Also:**
get motor pwm count(), set motor pwm event(), set motor unit(), set motor pwm duty()

## setup_msi( )

**Syntax:**
setup_msi***(settings)***;

**Parameters:**
***settings*** - Setting to setup the MSI peripheral in. The available options depend on whether the program is for the Master or Slave core of the device. See the device's header file for valid defines that can be used with the function.

**Returns:**
---

**Function:**
Sets up the Master Slave Interface (MSI).

**Availability:**
Only available on Dual Core devices.

**Requires:**
---

**Examples:**
```
//Setup MSI to enable the Slave PIC, interrupt when 1st write to
//FIFO is done by other core, enable write FIFO and enable read FIFO.
setup_msi(MSI_SLAVE_ENABLE | MSI_FIFO_DATA_VALID_INT_ON_1ST_WRITE |
MSI_WRITE_FIFO_ENABLED | MSI_READ_FIFO_ENABLED);
```

**See Also:**
msi_write_mailbox(), msi_read_mailbox(), msi_status(), msi_read_fifo(), msi_mailbox_status(),msi_write_fifo(), msi_fifo_status()

## setup_nco( )

**Syntax:**
setup_nco(**settings**,**inc_value**)

**Parameters:**
***settings*** - setup of the NCO module. See the device's .h file for all options. Some typical options include:

NCO_ENABLE
NCO_OUTPUT
NCO_PULSE_FREQ_MODE
NCO_FIXED_DUTY_MODE

***inc_value*** - value to increment the NCO 20 bit accumulator by.

**Returns:**
---

**Function:**
Sets up the NCO module and sets the value to increment the 20-bit accumulator by.

**Availability:**
Devices with a NCO module.

**Requires:**
---
**Examples:**
```
setup_nco(NCO_ENABLED|NCO_OUTPUT|NCO_FIXED_DUTY_MODE|NCO_CLOCK_F
OSC,8192);
```

**See Also:**
get_nco_accumulator( ), set_nco_inc_value( ), get_nco_inc_value( )

## setup_opamp1( ) setup_opamp2( ) setup_opamp3( ) setup_opamp4( )

**Syntax:**
setup_opamp1(***mode***)
setup_opamp2(***mode***)
setup_opamp3(***mode***)
setup_opamp4(**mode**)

**Parameters:**
***mode*** - The mode of the operation amplifier. See the devices .h file for all options.
Some typical options include:
OPAMP_ENABLED
OPAMP_DISABLED

**Returns:**
---

**Function:**
Enables or Disables the internal operational amplifier peripheral of certain devices.

**Availability:**
Devices with a built-in operational amplifier (for example, PIC16F785)

**Requires:**
---

**Examples:**
```
  setup_opamp1(OPAMP_ENABLED);
  setup_opamp2(OPAMP_DISABLED);
  setup_opamp3(OPAMP_ENABLED | OPAMP_I_TO_OUTPUT);
```

## setup_opamp1( ) setup_opamp2( ) setup_opamp3( ) setup_opamp4( )

**Syntax:**
setup_opamp1(***mode***)
setup_opamp2(***mode***)
setup_opamp3(***mode***)
setup_opamp4(**mode**)

**Parameters:**
***mode*** - The mode of the operation amplifier. See the devices .h file for all options.
Some typical options include:
OPAMP_ENABLED
OPAMP_DISABLED

**Returns:**
---

**Function:**
Enables or Disables the internal operational amplifier peripheral of certain devices.

**Availability:**
Devices with a built-in operational amplifier (for example, PIC16F785)

**Requires:**
---

**Examples:**
```
  setup_opamp1(OPAMP_ENABLED);
  setup_opamp2(OPAMP_DISABLED);
  setup_opamp3(OPAMP_ENABLED | OPAMP_I_TO_OUTPUT);
```

## setup_oscillator( )

**Syntax:**
setup_oscillator(***mode***, ***finetune***)

**Parameters:**
***mode -*** is dependent on the chip. For example, some chips allow speed setting such as OSC_8MHZ or OSC_32KHZ. Other chips permit changing the source like OSC_TIMER1.

***finetune -*** (only allowed on certain parts) is a signed int with a range of -31 to +31.

**Returns:**
Some devices return a state such as OSC_STATE_STABLE to indicate the oscillator is stable.

**Function:**
This function controls and returns the state of the internal RC oscillator on some parts. See the devices .h file for valid options for a particular device.

Note that if INTRC or INTRC_IO is specified in #fuses and a #USE DELAY is used for a valid speed option, then the compiler will do this setup automatically at the start of main().

***WARNING***: If the speed is changed at run time the compiler may not generate the correct delays for some built in functions. The last #USE DELAY encountered in the file is always assumed to be the correct speed. You can have multiple #USE DELAY lines to control the compilers knowledge about the speed.

**Availability:**
Devices with a OSCCON register.

**Requires:**
Constants are defined in the .h file.

**Examples:**
```
setup_oscillator( OSC_2MHZ );
```

**See Also:**
#FUSES, Internal oscillator Overview

***[PCD]***
**Syntax:**
setup_oscillator(***mode***, ***target [,source] [,divide]*** )

**Parameters:**
***mode -*** is one of:
OSC_INTERNAL
OSC_CRYSTAL
OSC_CLOCK
OSC_RC
OSC_SECONDARY

***target -*** is the target frequency to run the device it.

***source -*** is optional. It specifies the external crystal/oscillator frequency. If omitted the value from the last #USE DELAY is used. If mode is OSC_INTERNAL, source is an optional tune value for the internal oscillator for devices that support it. If omitted a tune value of zero will be used.

***divide in -*** is optional. For devices that support it, it specifies the divide ration for the Display Module Interface Clock. A number from 0 to 64 divides the clock from 1 to 17 increasing in increments of 0.25, a number from 64 to 96 divides the clock from 17 to 33 increasing in increments of 0.5, and a number from 96 to 127 divides the clock from 33 to 64 increasing in increments of 1. If omitted zero will be used for divide by 1.

**Returns:**
---

**Function:**
Configures the oscillator with preset internal and external source configurations. If the device fuses are set and #use delay() is specified, the compiler will configure the oscillator. Use this function for explicit configuration or programming dynamic clock switches. Please consult your target data sheets for valid configurations, especially when using the PLL multiplier, as many frequency range restrictions are specified.

**Availability:**
All Devices.

**Requires:**
Constants are defined in the .h file.

**Examples:**
```
setup_oscillator( OSC_CRYSTAL, 4000000, 16000000);
setup_oscillator( OSC_INTERNAL, 29480000);
```

***See Also:***
setup_wdt( ), Internal Oscillator Overview

## setup_pga( )

**Syntax:**
setup_pga(**module,settings**)

**Parameters:**
***module*** - constant specifying the Programmable Gain Amplifier (PGA) to setup.

**Returns:**
---

**Function:**
This function allows for setting up one of the Programmable Gain Amplifier modules.

**Availability:**
Devices with a Programmable Gain Amplifier module.

**Requires:**
---

**Examples:**

```
setup_pga(PGA_ENABLED | PGA_POS_INPUT_PGAxP1 | PGA_GAIN_8X);
```

## setup_pid( )

**Syntax:**
setup_pid([**mode**,[**K1**],[**K2**],[**K3**]);

**Parameters:**
***mode*** - the setup of the PID module. The options for setting up the module are defined in the device's header file as:

PID_MODE_PID
PID_MODE_SIGNED_ADD_MULTIPLY_WITH_ACCUMULATION
PID_MODE_SIGNED_ADD_MULTIPLY
PID_MODE_UNSIGNED_ADD_MULTIPLY_WITH_ACCUMULATION
PID_MODE_UNSIGNED_ADD_MULTIPLY
PID_OUTPUT_LEFT_JUSTIFIED
PID_OUTPUT_RIGHT_JUSTIFIED

***K1*** - optional parameter specifying the K1 coefficient, defaults to zero if not specified. The K1 coefficient is used in the PID and ADD_MULTIPLY modes. When in PID mode the K1 coefficient can be calculated with the following formula:

K1 = Kp + Ki * T + Kd/T

When in one of the ADD_MULTIPLY modes K1 is the multiple value.

***K2*** - optional parameter specifying the K2 coefficient, defaults to zero if not specified. The K2 coefficient is used in the PID mode only and is calculated with the following formula:

K2 = -(Kp + 2Kd/T)

***K3*** - optional parameter specifying the K3 coefficient, defaults to zero if not specified. The K3 coefficient is used in the PID mode, only and is calculated with the following formula:

K3 = Kd/T

***T -*** is the sampling period in the above formulas.

**Returns:**
---

**Function:**
Setup the Proportional Integral Derivative (PID) module, and to set the input coefficients (K1, K2 and K3).

**Availability:**
Devices with built in PID module

**Requires:**
Constants are defined in the device's .h file.

**Examples:**

```
setup_pid(PID_MODE_PID, 10, -3, 50);
```

**See Also:**
pid_get_result(), pid_read(), pid_write(), pid_busy()

## setup_pmp(option,address_mask)

**Syntax:**
setup_pmp(***options,address_mask***);

**Parameters:**
**options** - The mode of the Parallel Master Port that allows to set the Master Port mode, read-write strobe options and other functionality of the PMPort module. See the device's .h file for all options. Some typical options include:

PAR_PSP_AUTO_INC

PAR_CONTINUE_IN_IDLE
PAR_INTR_ON_RW // Interrupt on read write
PAR_INC_ADDR // Increment address by 1 every read/write cycle
PAR_MASTER_MODE_1 // Master Mode 1
PAR_WAITE4 // 4 Tcy Wait for data hold after strobe

**address_mask** - this allows the user to setup the address enable register with a 16-bit value. This value determines which address lines are active from the available 16 address lines PMA0:PMA15.

**Returns:**
---

**Function:**
Configures various options in the PMP module. The options are present in the device's .h file and they are used to setup the module. The PMP module is highly configurable and this function allows users to setup configurations like the Slave module, Interrupt options, address increment/decrement options, Address enable bits, and various strobe and delay options.

**Availability:**
Devices with built in Parallel Master Port module.

**Requires:**
Constants are defined in the device's .h file.

**Examples:**

```
setup_psp(PAR_ENABLE|          //Sets up Master mode with address
PAR_MASTER_MODE_1|PAR_         //lines PMA0:PMA7
STOP_IN_IDLE,0x00FF);
```

**See Also:**
setup_pmp( ), pmp_address( ), pmp_read( ), psp_read( ), psp_write( ), pmp_write( ), psp_output_full( ), psp_input_full( ), psp_overflow( ), pmp_output_full( ), pmp_input_full( , pmp_overflow( )

## setup_power_pwm( )

**Syntax:**
setup_power_pwm(***modes***, ***postscale***, ***time_base***, ***period***, ***compare***, ***compare_postscale***, ***dead_time***)

**Parameters:**
***modes -*** values may be up to one from each group of the following:

PWM_CLOCK_DIV_4, PWM_CLOCK_DIV_16,
PWM_CLOCK_DIV_64, PWM_CLOCK_DIV_128

PWM_DISABLED, PWM_FREE_RUN, PWM_SINGLE_SHOT,
PWM_UP_DOWN, PWM_UP_DOWN_INT

PWM_OVERRIDE_SYNC

PWM_UP_TRIGGER,

PWM_DOWN_TRIGGER
PWM_UPDATE_DISABLE, PWM_UPDATE_ENABLE

PWM_DEAD_CLOCK_DIV_2,
PWM_DEAD_CLOCK_DIV_4,
PWM_DEAD_CLOCK_DIV_8,
PWM_DEAD_CLOCK_DIV_16

***postscale -*** is an integer between 1 and 16. This value sets the PWM time base output postscale.

***time_base -*** is an integer between 0 and 65535. This is the initial value of the PWM base

***period -*** is an integer between 0 and 4095. The PWM time base is incremented until it reaches this number.

***compare -*** is an integer between 0 and 255. This is the value that the PWM time base is compared to, to determine if a special event should be triggered.

***compare_postscale -*** is an integer between 1 and 16. This postscaler affects compare, the special events trigger.

***dead_time -*** is an integer between 0 and 63. This value specifies the length of an off period that should be inserted between the going off of a pin and the going on of it is a complementary pin.

**Returns:**
---

**Function:**
Initializes and configures the motor control Pulse Width Modulation (PWM) module.

**Availability:**
Devices with motor control or power PWM module.

**Requires:**
---
**Examples:**

```
setup_power_pwm(PWM_CLOCK_DIV_4|PWM_FREE_RUN|PWM_DEAD_CLOCK_DIV_4,1,100
00,1000,0,1,0);
```

**See Also:**
set_power_pwm_override(), setup_power_pwm_pins(), set_power_pwmX_duty()

## setup_power_pwm_faults( )

**Syntax:**
setup_power_pwm_faults(***mode***);

**Parameters:**
***mode* -** to setup the Power PWM faults. Valid options vary by device. See the device's header file for all options.

**Returns:**
---

**Function:**
Used to setup the power PWM faults for the Power Control PWM module.

**Availability:**
Devices with a Power Control PWM module.

**Requires:**
---

**Examples:**

```
setup_power_pwm_faults(PWM_ENABLE_FLTA | PWM_AUTO_CLEAR_FLTA);
```

**See Also:**
set_power_pwm_override(), setup_power_pwm_pins(), set_power_pwmX_duty(), setup_power_pwm( )

## setup_power_pwm_pins( )

**Syntax:**
setup_power_pwm_pins(**module**0,**module**1,**module**2,**module**3)

**Parameters:**
For each module (two pins) specify:
PWM_PINS_DISABLED
PWM_ODD_ON
PWM_BOTH_ON'PWM_COMPLEMENTARY
**Returns:**
---

**Function:**
Configures the pins of the Pulse Width Modulation (PWM) device.

**Availability:**
Devices with motor control or power PWM module.

**Requires:**
---

**Examples:**

```
setup_power_pwm_pins(PWM_PINS_DISABLED, PWM_PINS_DISABLED,
PWM_PINS_DISABLED,
   PWM_PINS_DISABLED);
setup_power_pwm_pins(PWM_COMPLEMENTARY,
   PWM_COMPLEMENTARY, PWM_PINS_DISABLED, PWM_PINS_DISABLED);
```

**See Also:**
setup_power_pwm(), set_power_pwm_override(),set_power_pwmX_duty()

## setup_prgx()

**Syntax:**
setup_prg1(**mode**, **current**, **rising_source**, **falling_source**);
setup_prg2(**mode**, **current**, **rising_source**, **falling_source**);
setup_prg3(**mode**, **current**, **rising_source**, **falling_source**);
setup_prg4(**mode**, **current**, **rising_source**, **falling_source**);

**Parameters:**
**mode** - the mode to setup the PRGx module in. The valid options vary depending on the device. See the device's header file for all options.

**current** - the current source/sink setting to set the PRGx module to and can be a value from 0 to 31. When using a value from 0 to 15, the current is calculated as: 2+(**current** / 2) uA. When using a value from 16 to 31, the current is calculated as: 10+(**current** - 16) uA.

**rising_source** - used to set the rising timing source. The valid options vary depending on the device. See the device's header file for all options.

**falling_source** - used to set the falling timing source. The valid options vary depending on the device. See the device's header file for all options.

**Returns:**
---

**Function:**
Used to set the PRGx modules.

**Availability:**
Devices that have a Programmable Ramp Generator (PRG) module.

**Requires:**
---

**Examples:**

```
setup_prg1(PRG_ENABLED | PRG_INPUT_SOURCD_FVR, 16,
PRG_RISING_SOURCE_CCP1, PRG_RISING_SOURCE_CCP2);
```

**See Also:**
prgx_status( )

## setup_psmc( )

**Syntax:**
setup_psmc(***unit, mode, period, period_time, rising_edge, rise_time, falling_edge, fall_time***);

**Parameters:**
***unit -*** is the PSMC unit number 1-4

***mode -*** is one of:

PSMC_SINGLE
PSMC_PUSH_PULL
PSMC_BRIDGE_PUSH_PULL
PSMC_PULSE_SKIPPING
PSMC_ECCP_BRIDGE_REVERSE
PSMC_ECCP_BRIDGE_FORWARD
PSMC_VARIABLE_FREQ
PSMC_3_PHASE

For complementary outputs use a bar (|) and or in **PSMC_COMPLEMENTARY**

Normally the module is not started until the **psmc_pins()** call is made. To enable immediately or in **PSMC_ENABLE_NOW**.

***period -*** has three parts or'ed together. The clock source, the clock divisor and the events that can cause the period to start.

***Sources***:

PSMC_SOURCE_FOSC
PSMC_SOURCE_64MHZ
PSMC_SOURCE_CLK_PIN

***Divisors***:

PSMC_DIV_1
PSMC_DIV_2
PSMC_DIV_4
PSMC_DIV_8

***Events -*** Use any of the events listed below.

***period_time -*** is the duration the period lasts in ticks. A tick is the above clock source divided by the divisor.

***rising_edge -*** is any of the following events to trigger when the signal goes active.

***rise_time -*** is the time in ticks that the signal goes active (after the start of the period) if the event is SMC_EVENT_TIME, otherwise unused.

***falling_edg***e - is any of the following events to trigger when the signal goes inactive.

***fall_time -*** is the time in ticks that the signal goes inactive (after the start of the period) if the event is PSMC_EVENT_TIME, otherwise unused.

***Events***:

PSMC_EVENT_TIME

PSMC_EVENT_C1OUT
PSMC_EVENT_C2OUT
PSMC_EVENT_C3OUT
PSMC_EVENT_C4OUT
PSMC_EVENT_PIN_PIN

**Returns:**
---

**Function:**
Initializes a PSMC unit with the primary characteristics such as the type of PWM, the period, duty and various advanced triggers. Normally this call does not start the PSMC. It is expected all the setup functions be called and the psmc_pins() be called last to start the PSMC module. These two calls are all that are required for a simple PWM. The other functions may be used for advanced settings and to dynamically change the signal.

**Availability:**
Devices with built in PSMC module.

**Requires:**
---
**Examples:**

```
//Simple PWM, 10khz out on pin C0 assuming a 20mhz crystal
// Duty is initially set to 25%

setup_psmc(1, PSMC_SINGLE,PSMC_EVENT_TIME | PSMC_SOURCE_FOSC, 100,
              PSMC_EVENT_TIME, 0,PSMC_EVENT_TIME, 25);
   psmc_pins(1, PSMC_A);
```

**See Also:**
psmc_deadband(), psmc_sync(), psmc_blanking(), psmc_modulation(), psmc_shutdown(), psmc_duty(), psmc_freq_adjust(), psmc_pins()

## setup_psp(option,address_mask)

**Syntax:**
setup_psp (***options,address_mask***);
setup_psp(***options***);

**Parameters:**
***Option*** - The mode of the Parallel slave port. This allows to set the slave port mode, read-write strobe options and other functionality of the PMP/EPMP module. See the devices .h file for all options. Some typical options include:

PAR_PSP_AUTO_INC
PAR_CONTINUE_IN_IDLE
PAR_INTR_ON_RW // Interrupt on read write
PAR_INC_ADDR // Increment address by 1 every read/write cycle
PAR_WAITE4 // 4 Tcy Wait for data hold after strobe

***address_mask*** - This allows the user to setup the address enable register with a 16 bit or 32 bit (EPMP) value. This value determines which address lines are active from the available 16 address lines PMA0: PMA15 or 32 address lines PMAO:PMA31 (EPMP only)

**Returns:**
---

**Function:**
Configures various options in the PMP/EPMP module. The options are present in the device.h file and they are used to setup the module. The PMP/EPMP module is highly configurable and this function allows users to setup configurations like the Slave mode, Interrupt options, address increment/decrement options, Address enable bits and various strobe and delay options.

**Availability:**
Devices with Parallel Port module or Enhanced Parallel Master Port module.

**Requires:**
Constants are defined in the devices .h file.

**Examples:**

```
setup_psp(PAR_PSP_AUTO_INC|       //Sets up legacy slave mode with
PAR_STOP_IN_IDLE,0x00FF );        //read and write buffers auto
increment
```

**See Also:**
psp_output_full(), psp_input_full(), psp_overflow(),
*[PCD]* setup_pmp() , pmp_address() , pmp_read() , psp_read() , psp_write() , pmp_write() ,pmp_output_full() , pmp_input_full() , pmp_overflow()

## setup_pwm1( ) setup_pwm2( ) setup_pwm3( ) setup_pwm4( )

**Syntax:**
setup_pwm1(**settings**);
setup_pwm2(**settings**);
setup_pwm3(**settings**);
setup_pwm4(**settings**);

**Parameters:**
settings- setup of the PWM module. See the device's .h file for all options. Some typical options include:

PWM_ENABLED
PWM_OUTPUT
PWM_ACTIVE_LOW

**Returns:**
---

**Function:**
Initializes the Pulse Width Modulation (PWM) device.

**Availability:**
Devices with PWM module.

**Requires:**
---

**Examples:**

```
setup_pwm1(PWM_ENABLED|PWM_OUTPUT);
```

## setup_qei( )

**Syntax:**
setup_qei( ***options, filter, maxcount*** );
*[PCD]* setup_qei( [***unit***,]***options***, ***filter***, ***maxcount*** );

**Parameters:**
Options - The mode of the QEI module. See the devices .h file for all options. Some common options are:

QEI_MODE_X2
QEI_MODE_X4

***filter*** - This parameter is optional, the user can enable the digital filters and specify the clock divisor.

***maxcount*** - Specifies the value at which to reset the position counter.

*[PCD]* ***Options***- The mode of the QEI module. See the devices .h file for all options. Some common options are:

QEI_MODE_X2
QEI_TIMER_GATED
QEI_TIMER_DIV_BY_1

*[PCD]* ***filter*** - This parameter is optional and the user can specify the digital filter clock divisor.

*[PCD]* ***maxcount*** - This will specify the value at which to reset the position counter.

*[PCD]* ***unit*** - Optional unit number, defaults to 1.

**Returns:**

---

**Function:**

Configures the Quadrature Encoder Interface. Various settings like mode and filters can be setup.

**Availability:**

Devices with QEI module.

**Requires:**

---

**Examples:**

```
setup_qei(QEI_MODE_X2|QEI_RESET_WHEN_MAXCOUNT,
EI_FILTER_ENABLE_QEA|QEI_FILTER_DIV_2,0x1000);
```

*[PCD]*

```
 setup_qei(QEI_MODE_X2|QEI_TIMER_INTERNAL,QEI_FILTER_DIV_2,QEI_FORWARD)
;
```

**See Also:**

qei_set_count() , qei_get_count() , qei_status()

## setup_rtc( )

**Syntax:**

setup_rtc(***options, calibration***);

[PCD] setup_rtc(**options**, **period**, **stability_time, sample_time**); //RTCC with Timestamp

**Parameters:**
***Options***- The mode of the RTCC module. See the devices .h file for all options

***Calibration***- This parameter is optional and the user can specify an 8 bit value that will get written to the calibration configuration register.

[PCD] ***Period*** - RTCC with Timestamp, sets the period of the clock divider counter. Value should be set to achieve a period of 0.5 seconds.

[PCD] ***Stability_time*** - RTCC with Timestamp, sets the Power Control Stability Time (2-255). This parameter is optional.

[PCD] ***Sample_time*** - RTCC with Timestamp, sets the Power Control Sample Time Window (2-255). This parameter is optional.
**Returns:**
---

**Function:**
Configures the Real Time Clock and Calendar module. The module requires an external 32.768 kHz clock crystal for operation.

**Availability:**
Devices with RTCC module.

**Requires:**
---

**Examples:**

```
setup_rtc(RTC_ENABLE | RTC_OUTPUT SECONDS, 0x00);
                // Enable RTCC module with seconds clock and no
calibration

[PCD] setup_rtc(RTC_ENABLE|RTC_CLOCK_SOSC, 16383);
                // Enable RTCC with Timestamp module from an external
32.768Khz crystal
```

**See Also:**
rtc_read(), rtc_alarm_read(), rtc_alarm_write(), setup_rtc_alarm(), rtc_write(, setup_rtc()

## setup_rtc_alarm( )

**Syntax:**
setup_rtc_alarm(***options***, ***mask***, ***repeat***);

**Parameters:**
***options*** - The mode of the RTCC module. See the devices .h file for all options

***mask*** - specifies the alarm mask bits for the alarm configuration.

***repeat*** - Specifies the number of times the alarm will repeat. It can have a max value of 255.

**Returns:**
---

**Function:**
Configures the alarm of the RTCC module.

**Availability:**
Devices with RTCC module.

**Requires:**
---

**Examples:**

```
setup_rtc_alarm(RTC_ALARM_ENABLE, RTC_ALARM_HOUR, 3);
```

**See Also:**
rtc_read(), rtc_alarm_read(), rtc_alarm_write(), setup_rtc_alarm(), rtc_write(), setup_rtc()

## setup_sd_adc( )

**Syntax:**
setup_sd_adc**(settings1, settings 2, settings3)**;

**Parameters:**
***settings1*** - settings for the SD1CON1 register of the SD ADC module. See the device's *.h* file for all options. Some options include:

1 SDADC_ENABLED
2 SDADC_NO_HALT
3 SDADC_GAIN_1
4 SDADC_NO_DITHER

5 SDADC_SVDD_SVSS
6 SDADC_BW_NORMAL

***settings2*** - settings for the SD1CON2 register of the SD ADC module. See the device's *.h* file for all options. Some options include:

7 SDADC_CHOPPING_ENABLED
8 SDADC_INT_EVERY_SAMPLE
9 SDADC_RES_UPDATED_EVERY_INT
10 SDADC_NO_ROUNDING

***settings3*** - settings for the SD1CON3 register of the SD ADC module. See the device's *.h* file for all options. Some options include:

11 SDADC_CLOCK_DIV_1
12 SDADC_OSR_1024
13 SDADC_CLK_SYSTEM

**Returns:**
---

**Function:**
Setup the Sigma-Delta Analog to Digital Converter (SD ADC) module.

**Availability:**
Devices with SD ADC module.

**Requires:**
---

**Examples:**

```
setup_sd_adc(SDADC_ENABLED | SDADC_DITHER_LOW,SDADC_CHOPPING_ENABLED |
             SDADC_INT_EVERY_5TH_SAMPLE |SDADC_RES_UPDATED_EVERY_INT,
             SDADC_CLK_SYSTEM |SDADC_CLOCK_DIV_4);
```

**See Also:**
set_sd_adc_channel(), read_sd_adc(), set_sd_adc_calibration()

## [PCD] setup_sent( )

**Syntax:**
setup_sent(**module, settings, tick_time**);
setup_sent(**module, settings, tick_time, [frame_time]);**

**Parameters:**
***module*** - the SENT peripheral to setup, 1 or 2 for most devices.

***settings*** - the mode to setup the SENT peripheral in. Constants for setting up the peripheral are defined in the device's header file. See the device's header file for all the possible options.

***tick_time*** - the tick time to set the SENT peripheral to, value is a time in *us* from 3 to 90.

***fame_time*** - optional parameter unless peripheral is set-up for transmitter mode and ***sent_uses_pause_pulse*** is used in settings parameter. It is used to set the frame time in us of the message.

**Returns:**
---

**Function:**
Used to setup the Single-Edge Nibble Transmission (SENT) peripheral.

**Availability:**
Devices with a SENT peripheral.

**Requires:**
---

**Examples:**

```
//Setup SENT1 peripheral for asynchronous transmitter
//mode with HW CRC generation enabled, pause pulse
//enabled, and to send 6 data nibbles with a tick time
//90us and a frame time of 50ms.
setup_sent(1, SENT_MODE_TRANSMITTER_ASYNCHRONOUS | SENT_ENABLE_HW_CRC |
SENT_USES_PAUSE_PULSE | SENT_DATA_NIBBLES_6, 90, 50000);
```

**Example Files:**
ex_sent_transmitter.c, ex_sent_receiver.c

**See Also:**
sent_getd( ), sent_putd( ), sent_status( )

## setup_smtx( )

**Syntax:**
setup_smt1(**mode,[period]**);

setup_smt2(**mode,[period]**);

**Parameters:**
***mode*** - The setup of the SMT module. See the device's .h file for all options. Some typical options include:

SMT_ENABLED
SMT_MODE_TIMER
SMT_MODE_GATED_TIMER
SMT_MODE_PERIOD_DUTY_CYCLE_ACQ

***period*** - Optional parameter for specifying the overflow value of the SMT timer, defaults to maximum value if not specified.

**Returns:**
---

**Function:**
Configures the Signal Measurement Timer (SMT) module.

**Availability:**
Devices with SMT module.

**Requires:**
---

**Examples:**

```
setup_smt1(SMT_ENABLED | SMT_MODE_PERIOD_DUTY_CYCLE_ACQ|
           SMT_REPEAT_DATA_ACQ_MODE | SMT_CLK_FOSC);
```

**See Also:**
smtx_status(), stmx_start(), smtx_stop(), smtx_update(), smtx_reset_timer(), smtx_read(), smtx_write()

## setup_spi( ) setup_spi2( ) setup_spi3( ) setup_spi4( )

**Syntax:**
setup_spi(***mode***)
setup_spi2(***mode***)
setup_spi3(***mode***)
setup_spi4(***mode***)

**Parameters:**
***mode*** may be:

SPI_MASTER, SPI_SLAVE, SPI_SS_DISABLED

SPI_L_TO_H, SPI_H_TO_L
SPI_CLK_DIV_4, SPI_CLK_DIV_16,
SPI_CLK_DIV_64, SPI_CLK_T2
SPI_SAMPLE_AT_END, SPI_XMIT_L_TO_H
*[PCD]* SPI_MODE_16B, SPI_XMIT_L_TO_H

Constants from each group may be or'ed together with |

**Returns:**
---

**Function:**
Initializes the Serial Port Interface (SPI). This is used for 2 or 3 wire serial devices that follow a common clock/data protocol.
*[PCD]* Configures the hardware SPI™ module.
SPI_MASTER will configure the module as the bus master
SPI_SLAVE will configure the module as a slave on the SPI™ bus
SPI_SS_DISABLED will turn off the slave select pin so the slave module receives any transmission on the bus.
SPI_x_to_y will specify the clock edge on which to sample and transmit data
SPI_CLK_DIV_x will specify the divisor used to create the SCK clock from system clock.

**Availability:**
Devices with SPI hardware module.

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
setup_spi(spi_master |spi_l_to_h | spi_clk_div_16 );
setup_spi(SPI_MASTER | SPI_L_TO_H | SPI_DIV_BY_16);
```

**Example Files:**
ex_spi.c

**See Also:**
spi_write(), spi_read(), spi_data_is_in(), spi_set_txcnt(), SPI Overview

## setup_timerx( )

**Syntax:**
setup_timerX(***mode***)
setup_timerX**(mode,period)**

**Parameters:**

***mode -*** is a bit-field comprised of the following configuration constants:

TMR_DISABLED: Disables the timer operation.

TMR_INTERNAL: Enables the timer operation using the system clock. Without divisions, the timer will increment on every instruction cycle. On PCD, this is half the oscillator frequency.

TMR_EXTERNAL: Uses a clock source that is connected to the SOSCI/SOSCO pins

TMR_EXTERNAL_SYNC: Uses a clock source that is connected to the SOSCI/SOSCO pins. The timer will increment on the rising edge of the external clock which is synchronized to the internal clock phases. This mode is available only for Timer1.

TMR_EXTERNAL_RTC: Uses a low power clock source connected to the SOSCI/SOSCO pins; suitable for use as a real time clock. If this mode is used, the low power oscillator will be enabled by the setup_timer function. This mode is available only for Timer1.

TMR_DIV_BY_X: X is the number of input clock cycles to pass before the timer is incremented. X may be 1, 8, 64 or 256.

TMR_32_BIT: This configuration concatenates the timers into 32 bit mode. This constant should be used with timers 2, 4, 6 and 8 only.

Period is an optional 16 bit integer parameter that specifies the timer period. The default value is 0xFFFF.

**Returns:**

---

**Function:**

Sets up the timer specified by X (May be 1 – 9). X must be a valid timer on the target device.

**Availability:**

This function is available on all devices that have a valid timer X. Use getenv or refer to the target datasheet to determine which timers are valid.

**Requires:**

Constants are defined in the device's .h file

**Examples:**

```
/* setup a timer that increments every 64th instruction cycle with an
overflow period of 0xA010 */

setup_timer2(TMR_INTERNAL | TMR_DIV_BY_64, 0xA010);

/* Setup another timer as a 32-bit hybrid  with a period of 0xFFFFFFFF
and a interrupt that will be fired when that timer overflows*/

setup_timer4(TMR_32_BIT);              //use get_timer45() to get the
timer value
enable_interrupts(int_timer5);        //use the odd number timer for
the interrupt
```

**See Also:**
Timer Overview, setup_timerX(), get_timerXY(), set_timerX(), set_timerXY()

## setup_timerA( )

**Syntax:**
setup_timer_A (**mode**);

**Parameters:**
***mode*** values may be:
TA_OFF, TA_INTERNAL, TA_EXT_H_TO_L, TA_EXT_L_TO_H
TA_DIV_1, TA_DIV_2, TA_DIV_4, TA_DIV_8, TA_DIV_16, TA_DIV_32,
TA_DIV_64,  TA_DIV_128, TA_DIV_256

Constants from different groups may be or'ed together with |.

**Returns:**
---

**Function:**
Sets up Timer A.

**Availability:**
This function is only available on devices with Timer A hardware.

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
setup_timer_A(TA_OFF);
```

```
setup_timer_A(TA_INTERNAL | TA_DIV_256);
setup_timer_A(TA_EXT_L_TO_H | TA_DIV_1);
```

**See Also:**
get_timerA( ), set_timerA( ), TimerA Overview

## setup_timerB( )

**Syntax:**
setup_timer_B (**mode**);

**Parameters:**
***mode*** values may be:
TB_OFF, TB_INTERNAL, TB_EXT_H_TO_L, TB_EXT_L_TO_H
TB_DIV_1, TB_DIV_2, TB_DIV_4, TB_DIV_8, TB_DIV_16, TB_DIV_32, TB_DIV_64, TB_DIV_128, TB_DIV_256

Constants from different groups may be or'ed together with |.

**Returns:**
---

**Function:**
Sets up Timer B.

**Availability:**
This function is only available on devices with Timer B hardware.

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
setup_timer_B(TB_OFF);
setup_timer_B(TB_INTERNAL | TB_DIV_256);
setup_timer_B(TA_EXT_L_TO_H | TB_DIV_1);
```

**See Also:**
get_timerB( ), set_timerB( ), TimerB Overview

## setup_timer0( )

**Syntax:**
setup_timer_0 (**mode**);

**Parameters:**
***mode -*** constants defined in the device's .h file. Some typical defines are:
TO_INTERNAL
TO_EXT_L_TO_H
TO_EXT_H_TO_I
TO_DIV_2, TO_DIV_4
(See device's .h file for all possible defines.)

One constant may be used from each group or'ed together with the | operator.

**Returns:**
---

**Function:**
Sets up the timer 0 (aka RTCC).

**Availability:**
All Devices. (**WARNING**: On older PIC16 devices, set-up of the prescaler may undo the WDT prescaler)

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
setup_timer_0 (TO_INTERNAL|TO_DIV2);
```

**See Also:**
get_timer0(), set_timer0(), setup counters()

## setup_timer1( )

**Syntax:**
setup_timer_1 (**mode**);

**Parameters:**
***mode***
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

One constant may be used from each group or'ed together with the | operator.

**Returns:**
---

**Function:**
Initializes timer 1. The timer value may be read and written to using SET_TIMER1() and GET_TIMER1()Timer 1 is a 16 bit timer.

With an internal clock at 20mhz and with the T1_DIV_BY_8 mode, the timer will increment every 1.6us. It will overflow every 104.8576ms.

**Availability:**
Available only on devices with timer 1 hardware.

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
setup_timer_1 ( T1_DISABLED );
setup_timer_1 ( T1_INTERNAL | T1_DIV_BY_4 );
setup_timer_1 ( T1_INTERNAL | T1_DIV_BY_8 );
```

**See Also:**
get_timer1(), set_timer1() , Timer1 Overview

## setup_timer2( )

**Syntax:**
setup_timer_2 (***mode, period, postscale***);

**Parameters:**
***mode***
T2_DISABLED
T2_DIV_BY_1, T2_DIV_BY_4, T2_DIV_BY_16

***period -*** is a int 0-255 that determines when the clock value is reset

***postscale -*** is a number 1-16 that determines how many timer overflows before an interrupt: (1 means once, 2 means twice, an so on).

**Returns:**
---

**Function:**
Initializes timer 2. The mode specifies the clock divisor (from the oscillator clock). The timer value may be read and written to using GET_TIMER2() and SET_TIMER2(). 2 is a 8-bit counter/timer.

**Availability:**
Available only on devices with timer 2 hardware.

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
setup_timer_2 ( T2_DIV_BY_4, 0xc0, 2)   //at 20mhz, the timer will
increment
                                        //every 800ns will overflow
every 154.4us,
                                         //and will interrupt every
308.us
```

**See Also:**
get_timer2(), set_timer2() , Timer2 Overview

## setup_timer3( )

**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:**
---

**Function:**
Initializes timer 3 or 4.The mode specifies the clock divisor (from the oscillator clock). The timer value may be read and written to using GET_TIMER3() and SET_TIMER3(). Timer 3 is a 16 bit counter/timer.

**Availability:**
Available only on devices with timer 3 hardware.

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
setup_timer_3 (T3_INTERNAL | T3_DIV_BY_2);
```

**See Also:**
get_timer3(), set_timer3()

## setup_timer4( )

**Syntax:**
setup_timer_4 (***mode***);

**Parameters:**
***mode* -** may be one of:
T4_DISABLED, T4_DIV_BY_1, T4_DIV_BY_4, T4_DIV_BY_16

***period* -** is a int 0-255 that determines when the clock value is reset

***postscale* -** is a number 1-16 that determines how many timer overflows before an interrupt: (1 means once, 2 means twice, and so on).

**Returns:**
---

**Function:**
Initializes timer 4. The mode specifies the clock divisor (from the oscillator clock). The timer value may be read and written to using GET_TIMER4() and SET_TIMER4(). Timer 4 is a 8 bit counter/timer.

**Availability:**
Available only on devices with timer 4 hardware.

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
setup_timer_4 ( T4_DIV_BY_4, 0xc0, 2);   // At 20mhz, the timer will
increment
                                         // every 800ns,will overflow
every 153.6us,
```

```
                                        // and will interrupt every
  307.2us
```

**See Also:**
get_timer4(), set_timer4()

## setup_timer5( )

**Syntax:**
setup_timer_5 (***mode***);

**Parameters:**
***mode* -** may be one or two of the constants defined in the devices .h file.
T5_DISABLED, T5_INTERNAL, T5_EXTERNAL, or T5_EXTERNAL_SYNC
T5_DIV_BY_1, T5_DIV_BY_2, T5_DIV_BY_4, T5_DIV_BY_8
T5_ONE_SHOT, T5_DISABLE_SE_RESET, or T5_ENABLE_DURING_SLEEP

**Returns:**
---

**Function:**
Initializes timer 5. The mode specifies the clock divisor (from the oscillator clock). The timer value may be read and written to using GET_TIMER5() and SET_TIMER5(). Timer 5 is a 16 bit counter/timer.

**Availability:**
Available only on devices with timer 5 hardware.

**Requires:**
Constants are defined in the device's .h file

**Examples:**

```
  setup_timer_5 (T5_INTERNAL | T5_DIV_BY_2);
```

**See Also:**
get_timer5(), set_timer5(), Timer5 Overview

## setup_uart( )

**Syntax:**
setup_uart(***baud***, ***stream***)
setup_uart(***baud***)

setup_uart(***baud, stream, clock)***

**Parameters:**
***baud -*** is a constant representing the number of bits per second. A one or zero may also be passed to control the on/off status.

***Stream -*** is an optional stream identifier.

*Chips with the advanced UART may also use the following constants:*
UART_ADDRESS UART only accepts data with 9th bit=1
UART_DATA UART accepts all data

Chips with the EUART H/W may use the following constants:
UART_AUTODETECT Waits for 0x55 character and sets the UART baud rate to match.
UART_AUTODETECT_NOWAIT Same as above function, except returns before 0x55 is received. KBHIT() will be true when the match is made. A call to GETC() will clear the character.
UART_WAKEUP_ON_RDA Wakes PIC up out of sleep when RCV goes from high to low

***clock*** - If specified this is the clock rate this function should assume. The default comes from the #USE DELAY.

**Returns:**
---

**Function:**
Similar to SET_UART_SPEED. If 1 is passed as a parameter, the UART is turned on, and if 0 is passed, UART is turned off. If a BAUD rate is passed to it, the UART is also turned on, if not already on.

**Availability:**
Available only on devices with built in UART.

**Requires:**
#USE RS232

**Examples:**

```
setup_uart(9600);
setup_uart(9600, rsOut);
```

**See Also:**
#USE RS232, putc(), getc(), RS232 I/O Overview

# setup_vref( ) setup_vref2( )

**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.

*[PCD]* ***mode -*** is a bit-field comprised of the following constants:
VREF_DISABLED
VREF_LOW ( Vdd * value / 24)
VREF_HIGH ( Vdd * value / 32 + Vdd/4 )
VREF_ANALOG

**Returns:**
---

**Function:**
Establishes the voltage of the internal reference that may be used for analog compares and/or for output on pin A2.
*[PCD]* Configures the voltage reference circuit used by the voltage comparator.

The voltage reference circuit allows you to specify a reference voltage that the comparator module may use. You may use the Vdd and Vss voltages as your reference or you may specify VREF_ANALOG to use supplied Vdd and Vss. Voltages may also be tuned to specific values in steps, 0 through 15. That value must be or'ed to the configuration constants.

**Availability:**
This function is only available on devices with VREF hardware.
*[PCD]* Some devices, consult the device datasheet.

**Requires:**
*[PCD]* Constants are defined in the devices .h file

**Examples:**

```
setup_vref (VREF_HIGH | 6);
// At VDD=5, the voltage is 2.19V

[PCD] /* Use the 15th step on the course setting */
setup_vref(VREF_LOW | 14);
```

**Example Files:**
ex_comp.c

**See Also:**
Voltage Reference Overview

## setup_wdt( )

**Syntax:**
setup_wdt (***mode***)

**Parameters:**
Constants:

WDT_18MS
WDT_36MS
WDT_72MS
WDT_144MS
WDT_288MS
WDT_576MS
WDT_1152MS
WDT_2304MS

For some parts:

WDT_ON
WDT_OFF

.
[PCD] Mode is a bit-field comprised of the following constants:

WDT_ON
WDT_OFF

Specific Time Options vary between chips, some examples are:

WDT_2ms
WDT_64MS
WDT_1S
WDT_16S

**Returns:**
---

**Function:**
Setup-wdt is used to set the timer that is allowed between calls to restart-wdt () before the chip is reset. Some parts also allow the wdt to be enabled/disabled and to run time by this function. Some parts do not allow the time to be changed at run time. 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.

Note: For PCH parts and PCM parts with software controlled WDT, setup_wdt( ) would enable/disable watchdog timer only if NOWDT fuse is set. If WDT fuse is set, watchdog timer is always enabled.

Note: WDT_OFF should not be used with any other options.
**Warning**: Some chips share the same prescaller between the WDT and Timer0. In these cases a call to setup_wdt may disable the Timer0 prescaller.

*[PCD]* Configures the watchdog timer. The watchdog timer is used to monitor the software. If the software does not reset the watchdog timer before it overflows, the device is reset, preventing the device from hanging until a manual reset is initiated. The watchdog timer is derived from the slow internal timer.

**Availability:**
All Devices (**WARNING**: On older PIC16 devices, set-up of the prescaler may undo the timer0 prescaler)

**Requires:**
Constants are defined in the devices .h file

**Examples:**

```
#fuses WDT1, WDT           // PIC18 example, See restart_wdt for a
PIC18 example
main() {
   setup_wdt( WDT_18MS);
   while (TRUE) {
     restart_wdt();
     perform_activity();
     }
}
```

*[PCD]*
```
setup_wdt(WDT_ON);
```

**Example Files:**
[PCD] ex_wdt.c

**See Also:**
#FUSES , restart_wdt() , WDT or Watch Dog Timer Overview , Internal Oscillator Overview

## setup_zcd( )

**Syntax:**
setup_zdc**(mode)**;

**Parameters:**
**mode**- the setup of the ZDC module. The options for setting up the module include:
ZCD_ENABLED
ZCD_DISABLED
ZCD_INVERTED
ZCD_INT_L_TO_H
ZCD_INT_H_TO_L

**Returns:**
---

**Function:**
Set-up the Zero_Cross Detection (ZCD) module.

**Availability:**
Devices with a ZCD module.

**Requires:**
---

**Examples:**

```
setup_zcd(ZCD_ENABLE|ZCD_INT_H_TO_L);
```

**See Also:**
zcd_status()

## shift_left( )

**Syntax:**
shift_left (***address***, ***bytes***, ***value***)

**Parameters:**
***address -*** is a pointer to memory.

***bytes -*** is a count of the number of bytes to work with

***value -*** is a 0 to 1 to be shifted in.

**Returns:**
0 or 1 for the bit shifted out

**Function:**
Shifts a bit into an array or structure. The address may be an array identifier or an address to a structure (such as &data). Bit 0 of the lowest byte in RAM is treated as the LSB.

**Availability:**
All Devices.

**Requires:**
---

**Examples:**

```
byte buffer[3];
for(i=0; i<=24; ++i){                      // Wait for clock high
  while (!input(PIN_A2));
   shift_left(buffer,3,input(PIN_A3));    // Wait for clock low
   while (input(PIN_A2));
}                                         // reads 24 bits from pin
A3,each bit
                                         //is read on a low to high on
pin A2
```

**Example Files:**
ex_extee.c, 9356.c

**See Also:**
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:**
---

**Examples:**

```
                                        //reads 16 bits from pin A1,
each bit is read
                                    // on a low to high on pin A2
struct {
   byte time;
   byte command : 4;
   byte source  : 4;} msg;

for(i=0; i<=16; ++i) {
   while(!input(PIN_A2));
   shift_right(&msg,3,input(PIN_A1));
   while (input(PIN_A2)) ;}          // This shifts 8 bits out PIN_A0,
LSB first.
for(i=0;i<8;++i)
   output_bit(PIN_A0,shift_right(&data,1,0));
```

**Example Files:**
ex_extee.c, 9356.c

**See Also:**
shift_left(), rotate_right(), rotate_left()

## sleep( )

**Syntax:**
sleep**(mode)**

**Parameters:**
***mode*** - for most chips this is not used. Check the device header for special options on some chips.

*[PCD]* ***mode*** configures what sleep mode to enter, mode is optional. If mode is SLEEP_IDLE, the PIC will stop executing code but the peripherals will still be operational. If mode is SLEEP_FULL, the PIC will stop executing code and the peripherals will stop being clocked, peripherals that do not need a clock or are using an external clock will still be operational. SLEEP_FULL will reduce power consumption the most. If no parameter is specified, SLEEP_FULL will be used.

**Returns:**
---

**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:**
---

**Examples:**

```
SLEEP();
[PCD]
disable_interrupts(INT_GLOBAL);
enable_interrupt(INT_EXT);
clear_interrupt();
sleep(SLEEP_FULL);                //sleep until an INT_EXT interrupt
```

```
                                  //after INT_EXT wake-up,
                                //will resume operation from this point
```

**Example Files:**
ex_wakup.c

**See Also:**
reset cpu()

## sleep_ulpwu( )

**Syntax:**
sleep_ulpwu(***time***)

**Parameters:**
***time* -** specifies how long, in us, to charge the capacitor on the ultra-low power wakeup pin (by outputting a high on PIN_A0).

*[PCD]* ***time* -** specifies how long, in us, to charge the capacitor on the ultra-low power wakeup pin (by outputting a high on PIN_B0).

**Returns:**
---

**Function:**
Charges the ultra-low power wake-up capacitor on PIN_A0 for time microseconds, and then puts the PIC to sleep. The PIC will then wake-up on an 'Interrupt-on-Change' after the charge on the cap is lost.

*[PCD]* Charges the ultra-low power wake-up capacitor on PIN_B0 for time microseconds, and then puts the PIC to sleep. The PIC will then wake-up on an 'Interrupt-on-Change' after the charge on the cap is lost.

**Availability:**
Devices with Ultra Low Power Wake-Up.

**Requires:**
---

**Examples:**

```
while(TRUE)
{
    if (input(PIN_A1))     //PCD devices use (PIN_B0)
```

```
        //do something
    else
        sleep_ulpwu(10);  //cap will be charged for 10us,
                          //then goto sleep
}
```

**See Also:**
#USE DELAY

## smtx_read( )

**Syntax:**
value_smt1_read(**which**);
value_smt2_read(**which**);

**Parameters:**
**which -** Specifies which SMT registers to read. The following defines have been made in the device's header file to select which registers are read:

SMT_CAPTURED_PERIOD_REG
SMT_CAPTURED_PULSE_WIDTH_REG
SMT_TMR_REG
SMT_PERIOD_REG

**Returns:**
32-bit value

**Function:**
Read the Capture Period Registers, Capture Pulse Width Registers, Timer Registers or Period Registers of the Signal Measurement Timer module.

**Availability:**
Devices with SMT module.

**Requires:**
---

**Examples:**

```
unsigned int32 Period;
Period = smt1_read(SMT_CAPTURED_PERIOD_REG);
```

**See Also:**
smtx_status(), stmx_start(), smtx_stop(), smtx_update(), smtx_reset_timer(), setup_SMTx(), smtx_write()

## smtx_reset_timer( )

**Syntax:**
smt1_reset_timer();
smt2_reset_timer();

**Parameters:**
---

**Returns:**
---

**Function:**
Manually reset the Timer Register of the Signal Measurement Timer module.

**Availability:**
Devices with SMT module.

**Requires:**
---

**Examples:**

```
smt1_reset_timer();
```

**See Also:**
setup_smtx(), stmx_start(), smtx_stop(), smtx_update(), smtx_status(), smtx_read(), smtx_write()

## smtx_start( )

**Syntax:**
smt1_start();
smt2_start();

**Parameters:**
---

**Returns:**
---

**Function:**
Allow the Signal Measurement Timer (SMT) module start acquiring data.

**Availability:**
Devices with SMT module.

**Requires:**
---

**Examples:**

```
smt1_start();
```

**See Also:**
smtx_status(), setup_smtx(), smtx_stop(), smtx_update(), smtx_reset_timer(), smtx_read(), smtx_write()

## smtx_status( )

**Syntax:**
value = smt1_status();
value = smt2_status();

**Parameters:**
---

**Returns:**
The status of the SMT module.

**Function:**
Return the status of the Signal Measurement Timer (SMT) module.

**Availability:**
Devices with SMT module.

**Requires:**
---

**Examples:**

```
status = smt1_status();
```

**See Also:**
setup_smtx(), stmx_start(), smtx_stop(), smtx_update(), smtx_reset_timer(),smtx_read(), smtx_write()

## smtx_stop( )

**Syntax:**
smt1_stop();
smt2_stop();

**Parameters:**
---

**Returns:**
---

**Function:**
Configures the Signal Measurement Timer (SMT) module.

**Availability:**
Devices with SMT module.

**Requires:**
---

**Examples:**

```
smt1_stop();
```

**See Also:**
smtx_status(), stmx_start(), setup_smtx(), smtx_update(), smtx_reset_timer(), smtx_read(), smtx_write()

## smtx_write( )

**Syntax:**
smt1_write(**which,value**);
smt2_write(**which,value**);

**Parameters:**
***which* -** Specifies which SMT registers to write. The following defines have been made in the device's header file to select which registers are written:

SMT_TMR_REG
SMT_PERIOD_REG

***value*** - The 24-bit value to set the specified registers.

**Returns:**
---

**Function:**
Write the Timer Registers or Period Registers of the Signal Measurement Timer (SMT) module.

**Availability:**
Devices with SMT module.

**Requires:**
---

**Examples:**

```
smt1_write(SMT_PERIOD_REG, 0x100000000);
```

**See Also:**
smtx_status(), stmx_start(), setup_smtx(), smtx_update(), smtx_reset_timer(), smtx_read(), setup_smtx()

## smtx_update( )

**Syntax:**
smt1_update(**which**);
smt2_update(**which**);

**Parameters:**
***which* -** Specifies which capture registers to manually update. The following defines have been made in the device's header file to select which registers are updated:
SMT_CAPTURED_PERIOD_REG
SMT_CAPTURED_PULSE_WIDTH_REG

**Returns:**
---

**Function:**
Manually update the Capture Period Registers or the Capture Pulse Width Registers of the Signal Measurement Timer module.

**Availability:**
Devices with SMT module.

**Requires:**
---

**Examples:**

```
smt1_update(SMT_CAPTURED_PERIOD_REG);
```

**See Also:**
setup_smtx(), stmx_start(), smtx_stop(), smtx_status(), smtx_reset_timer(), smtx_read(), smtx_write()

## spi_data_is_in( ) spi_data_is_in2( )spi_data_is_in3( )

**spi_data_is_in4( )**

**Syntax:**
result = spi_data_is_in()
result = spi_data_is_in2()
result = spi_data_is_in3()
result = spi_data_is_in4()

**Parameters:**
---
**Returns:**
0 (FALSE) or 1 (TRUE)

**Function:**
Returns TRUE if data has been received over the SPI.

**Availability:**
Devices with SPI hardware.

**Requires:**
---

**Examples:**

```
spi_data_is_in() && input(PIN_B2) );
if( spi_data_is_in() )
       data = spi_read();
```

**See Also:**
spi_read(), spi_write(), spi_set_txcnt(), SPI Overview

## spi_init( )

**Syntax:**
spi_init(baud);
spi_init(stream,baud);

**Parameters:**
**stream** – is the SPI stream to use as defined in the STREAM=name option in #USE SPI.

**band** - the band rate to initialize the SPI module to. If FALSE it will disable the SPI module, if TRUE it will enable the SPI module to the baud rate specified in #use SPI.
**Returns:**
---

**Function:**
Initializes the SPI module to the settings specified in #USE SPI.

**Availability:**
Devices with SPI hardware.

**Requires:**
#USE SPI

**Examples:**

```
while #use spi(MATER, SPI1, baud-1000000, mode=0, stream=SPI1_MODE0)

spi_spi_init(SPI1_MODE0, TRUE);                  //initialize and
enable
                                                  //SPI1 to setting in
#USE SPI1
spi_spi_init(FALSE);                              //disable SPI1
spi_spi_init(250000);                            //initialize and
enable SPI1
                                                  //to a baud rate of
250K
```

**See Also:**
#USE SPI, spi_xfer(), spi_xfer_in(), spi_prewrite(), spi_speed()

## spi_prewrite( )

**Syntax:**
spi_prewrite(**data**);
spi_prewrite(**stream**, **data**);

**Parameters:**
**stream -** is the SPI stream to use as defined in the STREAM=name option in #USE SPI.

**data** - the variable or constant to transfer via SPI
**Returns:**
---

**Function:**
Writes data into the SPI buffer without waiting for transfer to be completed. Can be used in conjunction with spi_xfer() with no parameters to transfer more then 8 bits for PCM and PCH device, or more then 8 bits or 16 bits (XFER16 option) for PCD. Function is useful when using the SSP or SSP2 interrupt service routines for PCM and PCH device, or the SPIx interrupt service routines for PCD device.

**Availability:**
Devices with SPI hardware.

**Requires:**
#USE SPI

**Examples:**

```
spi_prewrite(data_out);
```

**Example Files:**
ex_spi.c

**See Also:**
#USE SPI, spi_xfer(), spi_xfer_in(), spi_init(), spi_speed()

## spi_read( ) spi_read2( ) spi_read3( ) spi_read4( )

*[PCD]* **spi_read_16( )**
*[PCD]* **spi_read2_16( )**
*[PCD]* **spi_read3_16( )**
*[PCD]* **spi_read4_16( )**
*[PCD]* **spi_read_32( )**
*[PCD]* **spi_read2_32( )**
*[PCD]* **spi_read3_32( )**
*[PCD]* **spi_read4_32( )**

**Syntax:**
value = spi_read([***data]***)

value = spi_read2([***data]***)
value = spi_read3([***data]***)
value = spi_read4([***data]***)
*[PCD]* value = spi_read_16([***data]***)
*[PCD]* value = spi_read2_16([***data]***)
*[PCD]* value = spi_read3_16([***data]***)
*[PCD]* value = spi_read4_16([***data]***)
*[PCD]* value = spi_read_32([***data]***)
*[PCD]* value = spi_read2_32([***data]***)
*[PCD]* value = spi_read3_32([***data]***)
*[PCD]* value = spi_read4_32([***data]***)

**Parameters:**
***data*** – optional parameter and if included is an 8 bit int.
*[PCD]* ***data*** – optional parameter and if included is an 16 bit or 32 bit int.
**Returns:**
An 8-bit int
*[PCD]* A 16-bit or 32-bit int.

**Function:**
Return a value read by the SPI. If a value is passed to the spi_read() the data will be clocked out and the data received will be returned. If no data is ready, spi_read() will wait for the data is a SLAVE or return the last DATA clocked in from spi_write().

If this device is the MASTER then either do a spi_write(data) followed by a spi_read() or do a spi_read(data). These both do the same thing and will generate a clock. If there is no data to send just do a spi_read(0) to get the clock.

If this device is a SLAVE then either call spi_read() to wait for the clock and data or use_spi_data_is_in() to determine if data is ready.

**Availability:**
Devices with SPI hardware.

**Requires:**
---

**Examples:**

```
data_in = spi_read(out_data);
```

**Example Files:**
ex_spi.c

**See Also:**
spi_write(), spi_data_is_in(), spi_set_txcnt(), SPI Overview

## spi_set_txcnt( )

**Syntax:**
spi_set_txcnt (***count***)

**Parameters:**
***count*** - int16 value indicating number of SPI transfers that SS1 pin will be driver to active level for.

**Returns:**
Undefined

**Function:**
Used to control the number of SPI transfers that the SS1 pin is driven to the active level for when SPI peripheral is setup as SPI Master. Once the value is written, the SS1 pin will be driver to the active state. Also requires that the ***#pin_select*** be used to assign a pin as the SS1 output pin.

**Availability:**
Only on PIC18 devices with a dedicated SPI peripheral.

**Requires:**
---

**Examples:**
```
  #pin_select SCK1OUT=PIN_C0
  #pin_select SDO1=PIN_C1
  #pin_select SDI1=PIN_C2
  #pin_select SS1OUT=PIN_C4

  setup_spi(SPI_MASTER|SPI_SCK_IDLE_LOW|SPI_XMIT_L_TO_H|
  SPI_CLK_FOSC,500000);

  spi_set_txcnt(3);
  spi_write(WRITE_COMMAND);
  spi_write(Address);
  spi_write(Data);
```

**See Also:**
setup_spi(), spi_write(), spi_read(), spi_data_is_in(), SPI Overivew

## spi_speed()

**Syntax:**
spi_speed(**baud**);
spi_speed(**stream**,**baud**);
spi_speed**(stream**,**baud**,**clock**);

**Parameters:**
**stream** – is the SPI stream to use as defined in the STREAM=name option in #USE SPI.

**band** - the band rate to set the SPI module to.

**clock** - the current clock rate to calculate the band rate with. If not specified it uses the value specified in #use delay().
**Returns:**
---

**Function:**
Sets the SPI module's baud rate to the specified value.

**Availability:**
Devices with SPI hardware.

**Requires:**
#USE SPI

**Examples:**

```
spi_speed(250000);
spi_speed(SPI1_MODE0, 250000);
spi_speed(SPI1_MODE0, 125000, 8000000);
```

**See Also:**
#USE SPI, spi_xfer(), spi_xfer_in(), spi_prewrite(), spi_init()

## [PCD] spi_transfer_write()

**Syntax:**
spi_transfer_write([**stream**], **data**, **count**);

**Parameters:**
**stream** – an optional parameter specifying the SPI stream to transfer the data with. Defaults to last used SPI stream in if not specified.

**data** - the pointer to an array of bytes to transfer via SPI. The pin used to transfer data is defined in the DO=option in #use SPI.

**clock** - the number of bytes to transfer via SPI.
**Returns:**
---

**Function:**
Used to transfer multiple bytes to an SPI device.

**Availability:**
All devices

**Requires:**
#USE SPI

**Examples:**

```
spi_transfer_write(Data, 128);
```

**See Also:**
#USE SPI, spi_xfer(),

## spi_write( ) spi_write2( ) spi_write3( ) spi_write4( )

*[PCD]* **spi_write_16( )**
*[PCD]* **spi_write2_16( )**
*[PCD]* **spi_write3_16( )**
*[PCD]* **spi_write4_16( )**
*[PCD]* **spi_write_32( )**
*[PCD]* **spi_write2_32( )**
*[PCD]* **spi_write3_32( )**
*[PCD]* **spi_write4_32( )**

**Syntax:**
spi_write([**wait**],***value***);
spi_write2(**[wait]**,***value***);
spi_write3(**[wait]**,**value**);
spi_write4(**[wait]**,**value**);

*[PCD]* spi_write_16([**wait**],***value***);
*[PCD]* spi_write2_16(**[wait]**,***value***);
*[PCD]* spi_write3_16(**[wait]**,**value**);
*[PCD]* spi_write4_16(**[wait]**,**value**);
*[PCD]* spi_write_32([**wait**],***value***);
*[PCD]* spi_write2_32(**[wait]**,***value***);
*[PCD]* spi_write3_32(**[wait]**,**value**);
*[PCD]* spi_write4_32(**[wait]**,**value**);

**Parameters:**
***value -*** is an 8 bit int
*[PCD]* ***value -*** is an 16 bit or 32 bit int

***wait***- an optional parameter specifying whether the function will wait for the SPI transfer to complete before exiting. Default is TRUE if not specified.
**Returns:**
---

**Function:**
Sends data out the SPI interface. This will cause clocks to be generated. This function will write the value out to the SPI. At the same time data is clocked out data is clocked in and stored in a receive buffer. The spi_read() may be used to read the buffer.

**Availability:**
Devices with SPI hardware.

**Requires:**
---

**Examples:**

```
spi_write( data_out );
data_in = spi_read();
```

**Example Files:**
ex_spi.c

**See Also:**
spi_read(), spi_data_is_in(), SPI Overview, spi_set_txcnt()

## spi_xfer( )

**Syntax:**
spi_xfer(data)
spi_xfer(stream, data)
spi_xfer(stream, data, bits)
result = spi_xfer(data)
result = spi_xfer(stream, data)
result = spi_xfer(stream, data, bits)

**Parameters:**
***data -*** is the variable or constant to transfer via SPI. The pin used to transfer data is defined in the DO=pin option in #USE SPI.

***stream -*** is the SPI stream to use as defined in the STREAM=name option in #USE SPI.

***bits -*** is how many bits of data will be transferred.
**Returns:**
The data read in from the SPI. The pin used to transfer result is defined in the DI=pin option in #USE SPI.

**Function:**
Transfers data to and reads data from an SPI device.

**Availability:**
Devices with SPI hardware.

**Requires:**
#USE SPI

**Examples:**

```
int i = 34;
spi_xfer(i);                    // transfers the number 34 via SPI
int trans = 34, res;
res = spi_xfer(trans);          // transfers the number 34 via SPI
                                // also reads the number coming in from
SPI
```

**See Also:**
#USE SPI

## spi_xfer_in( )

**Syntax:**
value = spi_xfer_in();
value = spi_xfer_in(bits);
value = spi_xfer_in(stream,bits);

**Parameters:**
***stream -*** is the SPI stream to use as defined in the STREAM=name option in #USE SPI.

***bits -*** is how many bits of data will be received.
**Returns:**
The data read in from the SPI.

**Function:**
Reads data from the SPI, without writing data into the transmit buffer first.

**Availability:**
Devices with SPI hardware.

**Requires:**
#USE SPI, and the option SLAVE is used in #USE SPI to setup PIC as a SPI slave device.

**Examples:**

```
data_in = spi_xfer_in();
```

**Example Files:**
ex_spi.c

**See Also:**
#USE SPI, spi_xfer(), spi_prewrite(), spi_init(), spi_speed()

## sprintf( )

**Syntax:**
sprintf(***string***, ***cstring***, ***values***...);
bytes=sprintf(***string***, ***cstring***, ***values***...)

**Parameters:**
***string -*** is an array of characters.

***cstring -*** is a constant string or an array of characters null terminated.

***values -*** are a list of variables separated by commas. Note that format specifies do not work in ram band strings.
**Returns:**
Bytes is the number of bytes written to string.

**Function:**
This function operates like printf() except that the output is placed into the specified string. The output string will be terminated with a null. No checking is done to ensure the string is large enough for the data. See printf() for details on formatting.

**Availability:**
All Devices

**Requires:**
---

**See Also:**
printf()

## sqrt( )

**Syntax:**
result = sqrt (***value***)

**Parameters:**
***value -*** is a float
*[PCD]* ***value -*** is any float type
**Returns:**
A float
*[PCD]* Returns a floating point value with a precision equal to **value**

**Function:**
Computes the non-negative square root of the float value x. If the argument is negative, the behavior is undefined.

Note on error handling:
If "errno.h" is included then the domain and range errors are stored in the errno variable. The user can check the errno to see if an error has occurred and print the error using the perror function.

Domain error occurs in the following cases:  sqrt: when the argument is negative

**Availability:**
All Devices

**Requires:**
#INCLUDE <math.h>

**Examples:**

```
distance = sqrt( pow((x1-x2),2)+pow((y1-y2),2) );
```

## srand( )

**Syntax:**
srand(***n***)

**Parameters:**
***n* -** is the seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to *rand*.

**Returns:**
---

**Function:**
The srand() function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand. If srand() is then called with same seed value, the sequence of random numbers shall be repeated. If rand is called before any call to srand() have been made, the same sequence shall be generated as when srand() is first called with a seed value of 1.

**Availability:**
All Devices

**Requires:**
#INCLUDE <STDLIB.H>

**Examples:**

```
srand(10);
I=rand();
```

**See Also:**
rand()

# STANDARD STRING FUNCTIONS( ) memchr( ) memcmp( ) strcat( ) strchr( ) strcmp( ) strcoll( ) strcspn( ) strerror( ) stricmp( ) strlen( ) strlwr( ) strncat( ) strncmp( ) strncpy( ) strpbrk( ) strrchr( ) strspn( ) strstr( ) strxfrm( )

**Syntax:**

| **ptr=strcat (*s1*, *s2*)** | **Concatenate s2 onto s1** |
|---|---|
| **ptr=strchr (*s1*, *c*)** | Find c in s1 and return &s1[i] |
| **ptr=strrchr (*s1*, *c*)** | Same but search in reverse |
| **cresult=strcmp (*s1*, *s2*)** | Compare s1 to s2 |
| **iresult=strncmp (*s1*, *s2*, *n*)** | Compare s1 to s2 (n bytes) |
| **iresult=stricmp (*s1*, *s2*)** | Compare and ignore case |
| **ptr=strncpy (*s1*, *s2*, *n*)** | Copy up to n characters s2->s1 |
| **iresult=strcspn (*s1*, *s2*)** | Count of initial chars in s1 not in s2 |
| **iresult=strspn (*s1*, *s2*)** | Count of initial chars in s1 also in s2 |
| **iresult=strlen (*s1*)** | Number of characters in s1 |
| **ptr=strlwr (*s1*)** | Convert string to lower case |
| **ptr=strpbrk (*s1*, *s2*)** | Search s1 for first char also in s2 |
| **ptr=strstr (*s1*, *s2*)** | Search for s2 in s1 |
| **ptr=strncat(*s1*,*s2*, *n*)** | Concatenates up to n bytes of s2 onto s1 |
| **iresult=strcoll(*s1*,*s2*)** | Compares s1 to s2, both interpreted as appropriate to the current locale. |
| **res=strxfrm(*s1*,*s2*,*n*)** | Transforms maximum of n characters of s2 and places them in s1, such that strcmp(s1,s2) will give the same result as strcoll(s1,s2) |
| **iresult=memcmp(*m1*,*m2*,*n*)** | Compare m1 to m2 (n bytes) |
| **ptr=memchr(*m1*,*c*,*n*)** | Find c in first n characters of m1 and return &m1[i] |
| **ptr=strerror(errnum)** | Maps the error number in errnum to an error message string. The parameters 'errnum' is an unsigned 8 bit int. Returns a pointer to the string. |

**Parameters:**
***s1*** and ***s2*** are pointers to an array of characters (or the name of an array).
Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi").

***n -*** is a count of the maximum number of character to operate on.

***c -*** is a 8 bit character

***m1*** and ***m2*** are pointers to memory.

**Returns & Functions:**
ptr is a copy of the s1 pointer
iresult is an 8 bit int
result is -1 (less than), 0 (equal) or 1 (greater than)
res is an integer.

**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

**See Also:**
strcpy(), strtok()

## strcpy( ) strcopy( )

**Syntax:**
strcpy (***dest***, ***src***)
strcopy (***dest***, ***src***)

**Parameters:**
***dest* -** is a pointer to a RAM array of characters.
***src* -** may be either a pointer to a RAM array of characters or it may be a constant string.

**Returns:**
---

**Function:**
Copies a constant or RAM string to a RAM string. Strings are terminated with a 0.

**Availability:**
All Devices

**Requires:**
---

**Examples:**

```
schar string[10], string2[10];
.
.
.
strcpy (string, "Hi There");

strcpy(string2,string);
```

**Example Files:**
ex_str.c

**See Also:**
strxxxx()

## strtod( )
## [PCD] strtof( ) [PCD] strto48( )

**Syntax:**
result=strtod(***nptr***,& ***endptr***)
*[PCD]* result=strtof(***nptr***,& ***endptr***)
*[PCD]* result=strtof48(***nptr***,& ***endptr***

**Parameters:**
***nptr*** and ***endptr*** are strings

**Returns:**
result is a float.
*[PCD]*
strtod returns a double precision floating point number.
strtof returns a single precision floating point number.
strtof48 returns a extended precision floating point number.

Returns the converted value in result, if any. If no conversion could be performed, zero is returned.

**Function:**
The strtod function converts the initial portion of the string pointed to by nptr to a float representation. The part of the string after conversion is stored in the object pointed to endptr, provided that endptr is not a null pointer. If nptr is empty or does not have the

expected form, no conversion is performed and the value of nptr is stored in the object pointed to by endptr, provided endptr is not a null pointer.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**

```
float result;                          //PCD devices, replace "float"
with "double"
char str[12]="123.45hello";
char *ptr;
result=strtod(str,&ptr);               //result is 123.45 and ptr is
"hello"
```

**See Also:**
strtol(), strtoul()

## strtok( )

**Syntax:**
ptr = strtok(***s1***, ***s2***)

**Parameters:**
***s1*** and ***s2*** are pointers to an array of characters (or the name of an array).
Note that s1 and s2 MAY NOT BE A CONSTANT (like "hi"). s1 may be 0 to indicate a continue operation.
**Returns:**
**ptr** points to a character in s1 or is 0

**Function:**
Finds next token in s1 delimited by a character from separator string s2 (which can be different from call to call), and returns pointer to it.

First call starts at beginning of s1 searching for the first character NOT contained in s2 and returns null if there is none are found.

If none are found, it is the start of first token (return value). Function then searches from there for a character contained in s2.

If none are found, current token extends to the end of s1, and subsequent searches for a token will return null.

If one is found, it is overwritten by '\0', which terminates current token. Function saves pointer to following character from which next search will start.

Each subsequent call, with 0 as first argument, starts searching from the saved pointer.

**Availability:**
All Devices

**Requires:**
#INCLUDE <string.h>

**Examples:**

```
char string[30], term[3], *ptr;

strcpy(string,"one,two,three;");
strcpy(term,",;");

ptr = strtok(string, term);
while(ptr!=0) {
   puts(ptr);
   ptr = strtok(0, term);
   }                              // Prints: one, two, three
```

**Example Files:**
ex_str.c

**See Also:**
strxxxx(), strcpy()

## strtol( )

**Syntax:**
result=strtol(***nptr***,& ***endptr***, ***base****)*

**Parameters:**
***nptr*** and ***endptr*** are strings and ***base*** is an integer
**Returns:**
Result is a signed long ***int***.
Returns the converted value in result , if any. If no conversion could be performed, zero is returned.

**Function:**
The ***strtol*** function converts the initial portion of the string pointed to by ***nptr*** to a signed long ***int*** representation in some radix determined by the value of base. The part of the string after conversion is stored in the object pointed to ***endptr***, provided that ***endptr*** is not a null pointer. If ***nptr*** is empty or does not have the expected form, no conversion is performed and the value of ***nptr*** is stored in the object pointed to by ***endptr***, provided ***endptr*** is not a null pointer.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**

```
signed long result;
char str[9]="123hello";
char *ptr;
result=strtol(str,&ptr,10);        //result is 123 and ptr is "hello
```

**See Also:**
strtod(), strtoul()

## strtoul( )

**Syntax:**
result=strtoul(***nptr***,***endptr***, ***base***)

**Parameters:**
***nptr*** and ***endptr*** are strings pointers and ***base*** is an integer 2-36.
**Returns:**
Result is a signed long ***int***.
Returns the converted value in result , if any. If no conversion could be performed, zero is returned.

**Function:**
The ***strtoul*** function converts the initial portion of the string pointed to by ***nptr*** to a long ***int*** representation in some radix determined by the value of base. The part of the string after conversion is stored in the object pointed to ***endptr***, provided that ***endptr*** is not a null pointer. If ***nptr*** is empty or does not have the expected form, no conversion is performed and the value of ***nptr*** is stored in the object pointed to by ***endptr***, provided ***endptr*** is not a null pointer.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdlib.h>

**Examples:**

```
long result;
char str[9]="123hello";
char *ptr;
result=strtoul(str,&ptr,10);       //result is 123 and ptr is "hello
```

**See Also:**
strtol(), strtod()

## swap( )

**Syntax:**
swap (***lvalue***)
*[PCD]* result = swap(***lvalue***)

**Parameters:**
***lvalue* -** is a byte variable
**Returns:**
undefined - WARNING: this function does not return the result
*[PCD]* A byte

**Function:**
Swaps the upper nibble with the lower nibble of the specified byte. This is the same as:
byte = (byte << 4) | (byte >> 4);

**Availability:**
All Devices

**Requires:**
---

**Examples:**

```
x=0x45;
swap(x);                //x now is 0x54
```

*[PCD]*

```
int x = 0x42;
int result;
```

```
result = swap(x);      // result is 0x24;
```

**See Also:**
rotate_right(), rotate_left()

## tolower( )  toupper( )

**Syntax:**
result = tolower (***cvalue***)
result = toupper (***cvalue***)

**Parameters:**
***cvalue* -** is a character
**Returns:**
An 8 bit character

**Function:**
These functions change the case of letters in the alphabet.

TOLOWER(X) will return 'a'..'z' for X in 'A'..'Z' and all other characters are unchanged.
TOUPPER(X) will return 'A'..'Z' for X in 'a'..'z' and all other characters are unchanged.

**Availability:**
All Devices

**Requires:**
---

**Examples:**

```
switch( toupper(getc()) ) {
   case 'R' : read_cmd();  break;
   case 'W' : write_cmd(); break;
   case 'Q' : done=TRUE;   break;
}
```

**Example Files:**
ex_str.c

## touchpad_getc( )

**Syntax:**
input = TOUCHPAD_GETC( );

**Parameters:**
---
**Returns:**
***char*** (returns corresponding ASCII number is “input” declared as ***int***)

**Function:**
Actively waits for firmware to signal that a pre-declared Capacitive Sensing Module (CSM) or charge time measurement unit (CTMU) pin is active, then stores the pre-declared character value of that pin in “input”.

Note: Until a CSM or CTMU pin is read by firmware as active, this instruction will cause the microcontroller to stall.

**Availability:**
Devices with CSM or CTMU Module.

**Requires:**
#USE TOUCHPAD (options)

**Examples:**

```
//When the pad connected to PIN_B0 is activated, store the letter 'A'

#USE TOUCHPAD (PIN_B0='A')
void main(void){
     char c;
     enable_interrupts(GLOBAL);

     c = TOUCHPAD_GETC();        //will wait until one of declared pins
is detected
                                 //if PIN_B0 is pressed, c will get
value 'A'
}
```

**See Also:**
#USE TOUCHPAD, touchpad_state( )

## touchpad_hit( )

**Syntax:**
value = TOUCHPAD_HIT( )

**Parameters:**
---
**Returns:**
TRUE or FALSE

**Function:**
Returns TRUE if a Capacitive Sensing Module (CSM) or Charge Time Measurement Unit (CTMU) key has been pressed. If TRUE, then a call to touchpad_getc() will not cause the program to wait for a key press.

**Availability:**
Devices with CSM or CTMU Module.

**Requires:**
#USE TOUCHPAD (options)

**Examples:**

```
//When the pad connected to PIN_B0 is activated, store the letter 'A'

#USE TOUCHPAD (PIN_B0='A')
void main(void){
     char c;
     enable_interrupts(GLOBAL);
     while (TRUE) {
     if ( TOUCHPAD_HIT() )               //wait until key on PIN_B0 is
pressed
     c = TOUCHPAD_GETC();                //get key that was pressed
     }                                   //c will get value 'A'
}
```

**See Also:**
#USE TOUCHPAD, touchpad_state( ), touchpad_getc( )

## touchpad_state( )

**Syntax:**
TOUCHPAD_STATE ***(state)***;

**Parameters:**
***state -*** is a literal 0, 1, or 2.
**Returns:**
---

**Function:**
Sets the current state of the touchpad connected to the Capacitive Sensing Module (CSM). The state can be one of the following three values:

0 : Normal state
1 : Calibrates, then enters normal state

2 : Test mode, data from each key is collected in the int16 array TOUCHDATA

Note: If the state is set to 1 while a key is being pressed, the touchpad will not calibrate properly.

**Availability:**
Devices with CSM or CTMU Module.

**Requires:**
#USE TOUCHPAD (options)

**Examples:**

```
#USE TOUCHPAD (THRESHOLD=5, PIN_D5='5', PIN_B0='C')
void main(void){
     char c;
     TOUCHPAD_STATE(1);             //calibrates, then enters normal
state
     enable_interrupts(GLOBAL);
     while(1){
           c = TOUCHPAD_GETC();     //will wait until one of declared pins
is detected
     }                              //if PIN_B0 is pressed, c will get
value 'C'
}                                   //if PIN_D5 is pressed, c will get
value '5'
```

**See Also:**
#USE TOUCHPAD, touchpad_getc( ), touchpad_hit( )

## tx_buffer_available( )

**Syntax:**
value = tx_buffer_available([**stream**]);

**Parameters:**
**stream** – optional parameter specifying the stream defined in #USE RS232.
**Returns:**
Number of bytes that can still be put into transmit buffer.

**Function:**
Function to determine the number of bytes that can still be put into transmit buffer before it overflows. Transmit buffer is implemented has a circular buffer, so be sure to check to make sure there is room for at least one more then what is actually needed.

**Availability:**
All Devices

**Requires:**
#USE RS232

**Examples:**

```
#USE_RS232(UART1,BAUD=9600,TRANSMIT_BUFFER=50)
void main(void) {
   unsigned int8 Count = 0;

   while(TRUE){
      if(tx_buffer_available()>13)
         printf("/r/nCount=%3u",Count++);
   }
}
```

**See Also:**
_USE_RS232( ), tx_buffer_full( ), rcv_buffer_bytes( ),rcv_buffer_full( ), get( ), putc( ) ,printf( ), setup_uart( ), putc_send( )

## tx_buffer_bytes( )

**Syntax:**
value = tx_buffer_bytes([**stream**]);

**Parameters:**
**stream** – optional parameter specifying the stream defined in #USE RS232.
**Returns:**
Number of bytes in transmit buffer that still need to be sent.

**Function:**
Function to determine the number of bytes in transmit buffer that still need to be sent.

**Availability:**
All Devices

**Requires:**
#USE RS232

**Examples:**

```
#USE_RS232(UART1,BAUD=9600,TRANSMIT_BUFFER=50)
```

```
void main(void) {
   char string[] = "Hello";
   if(tx_buffer_bytes() <= 45)
   printf("%s",string);
}
```

**See Also:**
_USE_RS232( ), tx_buffer_full( ), rcv_buffer_bytes( ), rcv_buffer_full( ), get( ), putc( ) ,printf( ), setup_uart( ), putc_send( )

## tx_buffer_full( )

**Syntax:**
value = tx_buffer_full([stream])

**Parameters:**
**stream** – optional parameter specifying the stream defined in #USE RS232
**Returns:**
TRUE if transmit buffer is full, FALSE otherwise.

**Function:**
Function to determine if there is room in transmit buffer for another character.

**Availability:**
All Devices

**Requires:**
#USE RS232

**Examples:**

```
#USE_RS232(UART1,BAUD=9600,TRANSMIT_BUFFER=50)
void main(void) {
   char c;
   if(!tx_buffer_full())
   putc(c);
}
```

**See Also:**
_USE_RS232( ), tx_buffer_bytes( ), rcv_buffer_bytes( ), rcv_buffer_full( ), get( ), putc( ) ,printf( ), setup_uart( ), putc_send( )

## va_arg( )

**Syntax:**
va_arg(**argptr**, **type)**

**Parameters:**
***argptr -*** is a special argument pointer of type va_list

***type -*** This is data type like int or char.
**Returns:**
The first call to va_arg after va_start return the value of the parameters after that specified by the last parameter. Successive invocations return the values of the remaining arguments in succession.

**Function:**
The function will return the next argument every time it is called.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdarg.h>

**Examples:**

```
int foo(int num, ...)
{
int sum = 0;
int i;
va_list argptr;                        // create special argument
pointer
va_start(argptr,num);                  // initialize argptr
for(i=0; i<num; i++)
   sum = sum + va_arg(argptr, int);
va_end(argptr);                       // end variable processing
return sum;
}
```

**See Also:**
nargs(), va_end(), va_start()

## va_end( )

**Syntax:**
va_end(**argptr**)

**Parameters:**
***argptr -*** is a special argument pointer of type va_list
**Returns:**
---

**Function:**
A call to the macro will end variable processing. This will facillitate a normal return from the function whose variable argument list was referred to by the expansion of va_start().

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdarg.h

**Examples:**

```
int foo(int num, ...)
{
int sum = 0;
int i;
va_list argptr;                  // create special argument pointer
va_start(argptr,num);            // initialize argptr
for(i=0; i<num; i++)
   sum = sum + va_arg(argptr, int);
va_end(argptr);                 // end variable processing
return sum;
}
```

**See Also:**
nargs(), va_start(), va_arg()

## va_start( )

**Syntax:**
va_start(**argptr**, **variable**)

**Parameters:**
***argptr -*** is a special argument pointer of type va_list

***variable*** – The second parameter to ***va_start()*** is the name of the last parameter before the variable-argument list.
**Returns:**
---

**Function:**
The function will initialize the ***argptr*** using a call to the macro ***va_start()***.

**Availability:**
All Devices

**Requires:**
#INCLUDE <stdarg.h

**Examples:**

```
int foo(int num, ...)
{
int sum = 0;
int i;
va_list argptr;                         // create special argument
pointer
va_start(argptr,num);                   // initialize argptr
for(i=0; i<num; i++)
   sum = sum + va_arg(argptr, int);
va_end(argptr);                         // end variable processing
return sum;
}
```

**See Also:**
nargs(), va_start(), va_arg()

## verify_slave_program( )

**Syntax:**
result = verify_slave_program(**address, instructions**);

**Parameters:**
***address*** - The address in the Master's program memory that the Slave program is stored at. Because of how the data is stored and written the address must be a multiple of 4.

***Instructions*** - The number of instructions to copy from the Master's Flash to the Slave's PRAM, because of how the data is written the number of instructions must be a multiple of 2.

**Returns:**
An int16 value indicating the number of errors, mis-matched instruction pairs, it found. If the return value is 0, then program loaded in the Slave's PRAM matches what is in the Master's Flash, i.e. the Slave program was verified.

**Function:**
Verify that the program loaded into the Slave's PRAM matched what is stored in the Master's Flash.

**Availability:**
Only available on Dual Core devices.

**Requires:**
---

**Examples:**
```
  Result = verify_slave_program(0x10000, 512);
  if(Result ==0)
       setup_msi_(MSI_SLAVE_ENABLED);
```

**See Also:**
load_slave_program()

## 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:**
---

**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:**
---

**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

## write_configuration_memory( )

**Syntax:**
write_configuration_memory (**[offset], dataptr,count**)

**Parameters:**
***dataptr -*** pointer to one or more bytes

***count -***: a 8 bit integer
***offset -*** is an optional parameter specifying the offset into configuration memory to start writing to, offset defaults to zero if not used.

**Returns:**
---

**Function:**
Erases all fuses and writes count bytes from the ***dataptr*** to the configuration memory. For Enhanced16 devices - erases and write User ID memory.

**Availability:**
All PIC18 Flash and Enhanced16 devices
All PIC24 Flash devices

**Requires:**
---

**Examples:**

```
int data[6];
write_configuration_memory(data,6
```

**See Also:**
WRITE_PROGRAM_MEMORY(), Configuration Memory Overview

## write_eeprom( )

**Syntax:**
write_eeprom (***address***, ***value***)
*[PCD]* write_eeprom ( ***address*** , ***pointer*** , ***N*** )

**Parameters:**
***address -*** is a (8 bit or 16 bit depending on the part) int, the range is device dependent

***value -*** is an 8 bit int

*[PCD]* ***address -*** is the 0 based starting location of the EEPROM write

*[PCD]* ***N -*** specifies the number of EEPROM bytes to write

*[PCD]* ***value -*** is a constant or variable to write to EEPROM

*[PCD]* ***pointer -*** is a pointer to location to data to be written to EEPROM

**Returns:**
---

**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 (like the 12CE671) see EX_EXTEE.c with CE51X.c, CE61X.c or CE67X.c.

*[PCD]* This function will write the specified value to the given address of EEPROM. If pointers are used than the function will write n bytes of data from the pointer to EEPROM starting at the value of address.
In order to allow interrupts to occur while using the write operation, use the #DEVICE option WRITE_EEPROM = NOINT. This will allow interrupts to occur while the write_eeprom() operations is polling the done bit to check if the write operations has completed. Can be used as long as no EEPROM operations are performed during an ISR.

**Availability:**
Devices with supporting hardware on chip.

**Requires:**
---

**Examples:**

```
#define LAST_VOLUME  10    // Location in EEPROM

volume++;
write_eeprom(LAST_VOLUME,volume);
```

**Example Files:**
ex_intee.c, ex_extee.c, ce51x.c, ce62x.c, ce67x.c

**See Also:**
read_eeprom(), erase_eeprom(), Data EEPROM Overview

## write_external_memory( )

**Syntax:**
write_external_memory( ***address***, ***dataptr***, ***count*** )

**Parameters:**
***address -*** is 16 bits on PCM parts and 32 bits on PCH parts

***dataptr -*** is a pointer to one or more bytes

***count -*** is a 8 bit integer

**Returns:**
---

**Function:**
Writes count bytes to program memory from ***dataptr*** to address. Unlike ***write_program_eeprom()*** and ***read_program_eeprom()*** this function does not use any special EEPROM/FLASH write algorithm. The data is simply copied from register address space to program memory address space. This is useful for external RAM or to implement an algorithm for external flash.

**Availability:**
PIC18 Devices Only

**Requires:**
---

**Examples:**

```
for(i=0x1000;i<=0x1fff;i++) {
   value=read_adc();
   write_external_memory(i, value, 2);
   delay_ms(1000);
}
```

**Example Files:**
ex_load.c, loader.c

**See Also:**
write_program_eeprom(), erase_program_eeprom(), Program Eeprom Overview

## write_extended_ram( )

**Syntax:**
write_extended_ram (**page,address,data,count**);

**Parameters:**
***page*** – the page in extended RAM to write to

***address*** – the address on the selected page to start writing to

***data*** – pointer to the data to be written

***count*** – the number of bytes to write (0-32768)

**Returns:**
---

**Function:**
Write data to the extended RAM of the microcontroller.

**Availability:**
Devices with more then 30K of RAM.

**Requires:**
---

**Examples:**

```
unsigned int8 data[8] = {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};

write_extended_ram(1,0x0000,data,8);
```

**See Also:**
read_extended_ram(), Extended RAM Overview

## write_program_eeprom( )

**Syntax:**
write_program_eeprom (***address***, ***data***)

**Parameters:**
***address -*** is 16 bits on PCM parts and 32 bits on PCH parts, data is 16 bits. The least significant bit should always be 0 in PCH.

**Returns:**
---

**Function:**
Writes to the specified program EEPROM area.

See write_program_memory() for more information on this function.

**Availability:**
Devices that allow writes to program memory.

**Requires:**
---

**Examples:**

```
write_program_eeprom(0,0x2800);    //disables program
```

**Example Files:**
ex_load.c, loader.c

**See Also:**
read_program_eeprom(), read_eeprom(), write_eeprom(), write_program_memory(), erase_program_eeprom(), Program Eeprom Overview

## write_program_memory( )

**Syntax:**
write_program_memory( ***address***, ***dataptr***, ***count*** );

**Parameters:**
***address -*** is 16 bits on PCM parts and 32 bits on PCH parts
*[PCD]* a***ddress -*** is 32 bits

***dataptr -*** is a pointer to one or more bytes

***count -*** is a 8 bit integer on PIC16 and 16-bit for PIC18
*[PCD]* ***count -*** is a 16 bit integer

**Returns:**
---

**Function:**
Writes count bytes to program memory from dataptr to address. This function is most effective when count is a multiple of FLASH_WRITE_SIZE. Whenever this function is about to write to a location that is a multiple of FLASH_ERASE_SIZE then an erase is performed on the whole block.

NOTES:Clarification about the functions to write to program memory:
In order to get the desired results while using write_program_memory(), the block of memory being written to needs to first be read in order to save any other variables currently stored there, then erased to clear all values in the block before the new values can be written. This is because the write_program_memory() function does not save any values in memory and will only erase the block if the first location is written to. If this process is not followed, when new values are written to the block, they will appear as garbage values.

For chips where getenv("FLASH_ERASE_SIZE") > getenv("FLASH_WRITE_SIZE")

write_program_eeprom() - Writes 2 bytes, does not erase (use erase_program_eeprom())

write_program_memory() - Writes any number of bytes, will erase a block whenever the first (lowest) byte in a block is written to. If the first address is not the start of a block that block is not erased.

erase_program_eeprom() - Will erase a block. The lowest address bits are not used.

For chips where getenv("FLASH_ERASE_SIZE") = getenv("FLASH_WRITE_SIZE")

write_program_eeprom() - Writes 2 bytes, no erase is needed.

write_program_memory() - Writes any number of bytes, bytes outside the range of the write block are not changed. No erase is needed.

erase_program_eeprom() - Not available

*[PCD]* Writes count bytes to program memory from dataptr to address. This function is most effective when count is a multiple of FLASH_WRITE_SIZE, but count needs to be a multiple of MIN_FLASH_WRITE. Whenever this function is about to write to a location that is a multiple of FLASH_ERASE_SIZE then an erase is performed on the whole block. Due to the 24 bit instruction length on PCD parts, every fourth byte of data is ignored. Fill the ignored bytes with 0x00.

See Program EEPROM Overview for more information on program memory access

**Availability:**
Devices that allow writes to program memory.

**Requires:**
---

**Examples:**

```
wfor(i=0x1000;i<=0x1fff;i++) {
   value=read_adc();
   write_program_memory(i, value, 2);
   delay_ms(1000);
}

[PCD]
for(i=0x1000;i<=0x1fff;i++) {
   value=read_adc();
   write_program_memory(i, &value, 4);
   delay_ms(1000);

int8 write_data[4] = {0x10,0x20,0x30,0x00};
write_program_memory (0x2000, write_data, 4);
```

**Example Files:**
loader.c

**See Also:**
write_program_eepro, erase_program_eeprom, Program Eeprom Overview

## write_program_memory8( )

**Syntax:**
write_PROGRAM_MEMORY8 (***address***, ***dataptr***, ***count*** );

**Parameters:**
***address*** is 16 bits to start writing data to the program memory.

***dataptr*** is a pointer to an array of bytes containing data to write to program memory.

***count*** is the number of bytes to write to program memory.

**Returns:**
Undefined

**Function:**
Write ***count*** bytes to program memory. This function only writes the least significant byte to each address in program memory. See **write_program_memory()** for a function that can write all the data to each address in program memory.

**Availability:**
Only on PCM devices with the ability to Read program memory.

**Requires:**
---

**Examples:**

```
write_program_memory8(Address, Data, 128);
```

**See Also:**
read_program_memory( ), write program memory( ), read_program_memory8( ), Program Eeprom Overview

## zcd_status( )

**Syntax:**
value=zcd_status()

**Parameters:**

**Returns:**
value - the status of the ZCD module. The following defines are made in the device's header file and are as follows:

ZCD_IS_SINKING

ZCD_IS_SOURCING

**Function:**
Determine if the Zero-Cross Detection (ZCD) module is currently sinking or sourcing current.
If the ZCD module is setup to have the output polarity inverted, the value return will be reversed.

**Availability:**
All devices with a ZCD module.

**Requires:**
---

**Examples:**

```
value=zcd_status():
```

**See Also:**
setup_zcd()

# STANDARD C INCLUDE FILES

## errno.h

| | |
|---|---|
| EDOM | Domain error value |
| ERANGE | Range error value |
| errno | error value |

## 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 *[PCD]* double significant |
| DBL_DIG: | Number of decimal digits, q, such that any floating point number or *[PCD]* double number with q decimal digits can be rounded into a floating point number or *[PCD]* double number with p radix b digits and back again without change to the q decimal digits. |
| DBL_MIN_EXP: | Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating point number or *[PCD]* double number. |
| DBL_MIN_10_EXP: | Minimum negative integer such that 10 raised to that power is in the range of normalized floating point number or *[PCD]* double number. |
| DBL_MAX_EXP: | Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating point number or *[PCD]* double number. |

| | |
|---|---|
| DBL_MAX_10_EXP: | Maximum negative integer such that 10 raised to that power is in the range of representable finite floating point number or *[PCD]* double number. |
| 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 or *[PCD]* double number. |
| LDBL_MANT_DIG: | Number of base digits in the floating point significant |
| LDBL_DIG: | Number of decimal digits, q, such that any floating point number with q decimal digits can be rounded into a floating point number with p radix b digits and back again without change to the q decimal digits. |
| LDBL_MIN_EXP: | Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number. |
| LDBL_MIN_10_EXP: | Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers. |
| LDBL_MAX_EXP: | Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number. |
| LDBL_MAX_10_EXP: | Maximum negative integer such that 10 raised to that power is in the range of representable finite floating-point numbers. |
| LDBL_MAX: | Maximum representable finite floating point number. |
| LDBL_EPSILON: | The difference between 1 and the least value greater than 1 that is representable in the given floating point type. |
| LDBL_MIN: | Minimum normalized positive floating point number. |

## limits.h

| | |
|---|---|
| CHAR_BIT: | Number of bits for the smallest object that is not a bit_field. |
| SCHAR_MIN: | Minimum value for an object of type signed char |
| SCHAR_MAX: | Maximum value for an object of type signed char |
| UCHAR_MAX: | Maximum value for an object of type unsigned char |
| CHAR_MIN: | Minimum value for an object of type char(unsigned) |
| CHAR_MAX: | Maximum value for an object of type char(unsigned) |
| MB_LEN_MAX: | Maximum number of bytes in a multibyte character. |
| SHRT_MIN: | Minimum value for an object of type short int |
| SHRT_MAX: | Maximum value for an object of type short int |
| USHRT_MAX: | Maximum value for an object of type unsigned short int |
| INT_MIN: | Minimum value for an object of type signed int |
| INT_MAX: | Maximum value for an object of type signed int |
| UINT_MAX: | Maximum value for an object of type unsigned int |
| LONG_MIN: | Minimum value for an object of type signed long int |
| LONG_MAX: | Maximum value for an object of type signed long int |

| | |
|---|---|
| ULONG_MAX: | Maximum value for an object of type unsigned long int |

## locale.h

| | |
|---|---|
| locale.h | (Localization not supported) |
| lconv | localization structure |
| SETLOCALE() | returns null |
| LOCALCONV() | returns clocale |

## setjmp.h

| | |
|---|---|
| jmp_buf: | An array used by the following functions |
| setjmp: | Marks a return point for the next longjmp |
| longjmp: | Jumps to the last marked point |

## stddef.h

| | |
|---|---|
| ptrdiff_t: | The basic type of a pointer |
| size_t: | The type of the sizeof operator (int) |
| wchar_t | The type of the largest character set supported (char) (8 bits) |
| NULL | A null pointer (0) |

## stdio.h

| | |
|---|---|
| stderr | The standard error s stream (USE RS232 specified as stream or the first USE RS232) |
| stdout | The standard output stream (USE RS232 specified as stream last USE RS232) |
| stdin | The standard input s stream (USE RS232 specified as stream last USE RS232) |

## stdlib.h

| | |
|---|---|
| div_t | structure type that contains two signed integers (quot and rem). |
| ldiv_t | structure type that contains two signed longs (quot and rem |
| EXIT_FAILURE | returns 1 |
| EXIT_SUCCESS | returns 0 |

| | |
|---|---|
| RAND_MAX- | |
| MBCUR_MAX- | 1 |
| SYSTEM() | Returns 0( not supported) |
| Multibyte character and string functions: | Multibyte characters not supported |
| MBLEN() | Returns the length of the string. |
| MBTOWC() | Returns 1. |
| WCTOMB() | Returns 1. |
| MBSTOWCS() | Returns length of string. |
| WBSTOMBS() | Returns length of string. |

Stdlib.h functions included just for compliance with ANSI C.

# SOFTWARE LICENSE AGREEMENT

**Carefully read this Agreement prior to opening this package. By opening this package, you agree to abide by the following provisions.**
**If you choose not to accept these provisions, promptly return the unopened package for a refund.**

All materials supplied herein are owned by Custom Computer Services, Inc. ("CCS") and is protected by copyright law and international copyright treaty. Software shall include, but not limited to, associated media, printed materials, and electronic documentation.

These license terms are an agreement between You ("Licensee" ) and CCS for use of the Software ("Software"). By installation, copy, download, or otherwise use of the Software, you agree to be bound by all the provisions of this License Agreement.

1. **LICENSE -** CCS grants Licensee a license to use in one of the two following options:
   1) Software may be used solely by single-user on multiple computer systems;
   2) Software may be installed on single-computer system for use by multiple users. Use of Software by additional users or on a network requires payment of additional fees.

   Licensee may transfer the Software and license to a third party; and such third party will be held to the terms of this Agreement. All copies of Software must be transferred to the third party or destroyed. Written notification must be sent to CCS for the transfer to be valid.

2. **APPLICATIONS SOFTWARE -** Use of this Software and derivative programs created by Licensee shall be identified as Applications Software, are not subject to this Agreement. Royalties are not be associated with derivative programs.

3. **WARRANTY -** CCS warrants the media to be free from defects in material and workmanship, and that the Software will substantially conform to the related documentation for a period of thirty (30) days after the date of purchase. CCS does not warrant that the Software will be free from error or will meet your specific requirements. If a breach in warranty has occurred, CCS will refund the purchase price or substitution of Software without the defect.

4. **LIMITATION OF LIABILITY AND DISCLAIMER OF WARRANTIES –** CCS and its suppliers disclaim any expressed warranties (other than the warranty contained in Section 3 herein), all implied warranties, including, but not limited to, the implied warranties of merchantability, of satisfactory quality, and of fitness for a particular purpose, regarding the Software.

   Neither CCS, nor its suppliers, will be liable for personal injury, or any incidental, special, indirect or consequential damages whatsoever, including, without limitation, damages for loss of profits, loss of data, business interruption, or any other commercial damages or losses, arising out of or related to your use or inability to use the Software.

   Licensee is responsible for determining whether Software is suitable for Applications.


PO BOX 2452
BROOKFIELD, WI 53008 U.S.A.

# Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

CCS:
52202-588