University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334

9

Experiment 9: Using HI-TECH C Compiler in MPLAB

Objectives

The main objectives of this experiment are to familiarize you with:

- Writing PIC programs in C
- Setting up MPLAB IDE projects to use the HI-TECH C compiler
- Becoming familiar with HI-TECH C primitives, built-in function in use with 10/12/16 MCU Family

Prepared by Eng. Enas Ja'ra

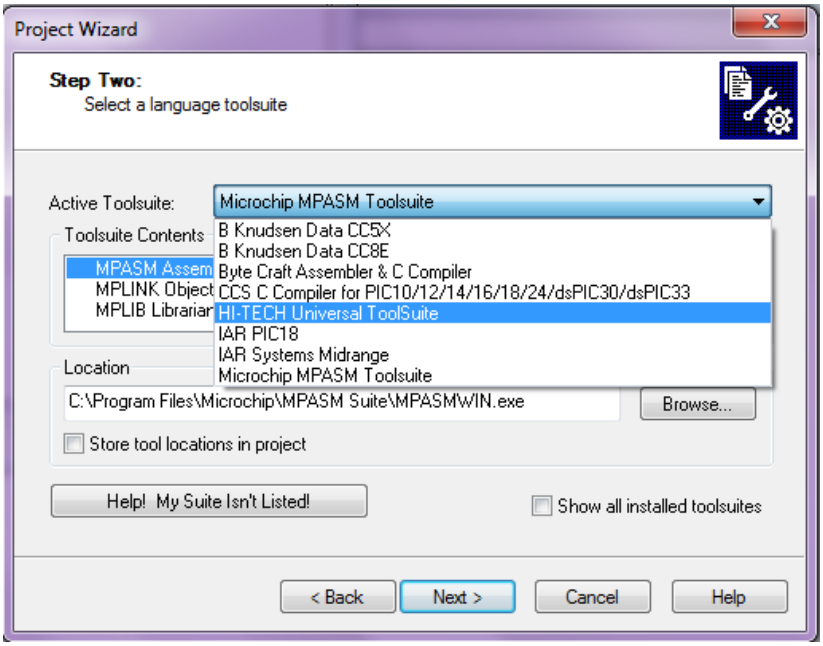
INTRODUCTION

So far in this lab course, PIC assembly programming has been introduced, however, in practice, most of the industrial and control codes are written in High Level Languages (abbreviated as HLL) the most common of which is the C programming language. The use of high level languages is preferred due to their simplicity which allows for faster program development (especially for large and very complex programs), easier debugging, and for easier future code maintainability, this will provide developers with shorter time to market advantages in a world where competition is at its prime to introduce new commercial products. On the other hand, HLLs assembled codes are often longer (due to inefficient compilers, aggressive and advanced optimizing compilers are often used to yield better results). Longer codes are at a disadvantage since memory space is limited in microcontrollers not to mention that longer codes take more time to execute. Expert assembly programmers can rewrite certain pieces of code in a very optimized and short fashion such that they execute faster, this is very important especially when real time applications are concerned. This direct use of assembly language requires that the programmer knows the problem in hand very well and that one is experienced in both software and target microcontroller hardwrae limitations. Often, programmers combine in between the use of C and Assembly language in the same developed source code.

There are many C compilers available commercially, such as mikroC, CCS and HI-TECH among others. This experiment introduces the "free" Lite version C compiler from HI-TECH software bundled with MPLAB, in contrast to the Pro versions of compilers commercially available from HI-TECH and others, the compiler and assembler don't use aggressive techniques and the resultant assembly codes are larger in size.

THIS PART ASSUMES YOU HAVE ALREADY SAVED A FILE WITH A C EXTENSION AND YOU HAVE ALREADY INSTALLED THE HI-TECH C PRO FOR THE PIC10/12/16 MCU FAMILY COMPILER

Create a project in MPLAB in the same steps as was shown in Exp 0, the only difference is in the step of selecting a language toolsuite; " Active Toolsuite" dialog box:

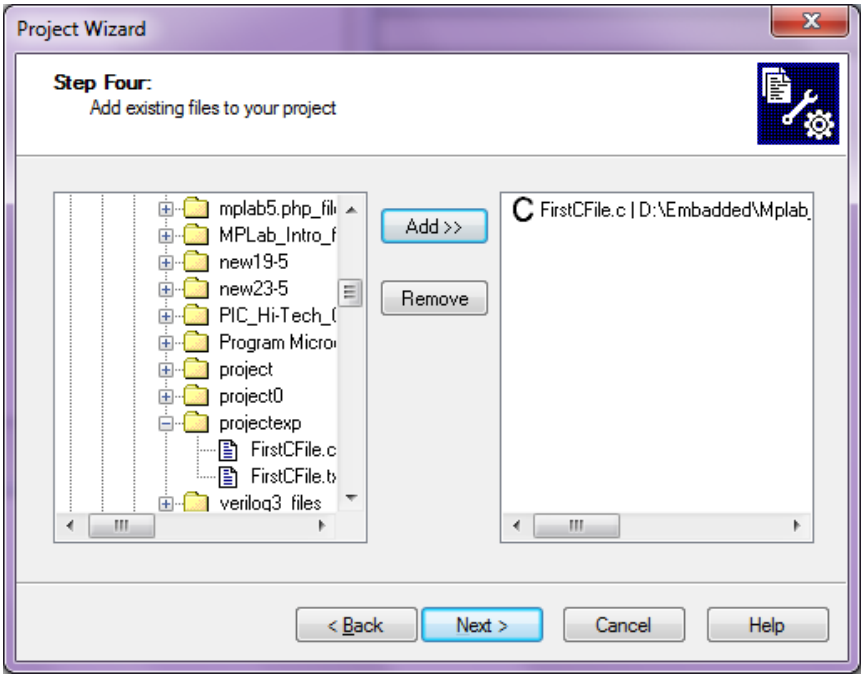


In this step where you get to specify the toolsuite associated with the project, you are not associating the project with the MPASM compiler as previously done, but instead we will be using the HI-TECH C compilers for Microchip devices

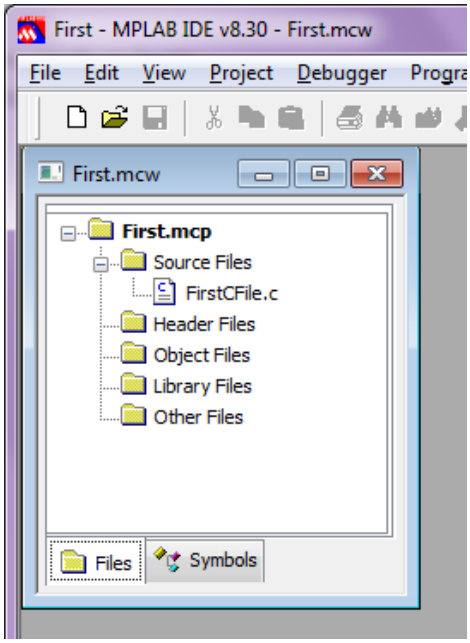
In the Active Toolsuite drop down menu, select HI-TECH Universal Toolsuite → Click next.

The next steps will proceed as usual:

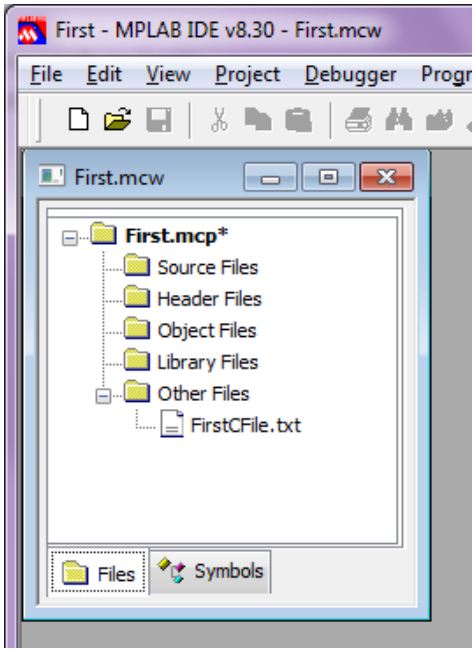
Browse to the directory where you saved your C file. Give your project a name → Save → Next. If you navigated correctly to your file destination you should see it in the left pane otherwise choose back and browse to the correct path. When done Click add your file to the project (here: FirstCFile.c). Make sure that the letter C is beside your file and not any other letter → Click next →Click Finish.



As before, you should see your C file under *Source file* list, now you are ready to begin.
Double click on the FirstCFile.C file in the project file tree to open. This is where you will write your programs, debug and simulate them.



CORRECT



WRONG

The proceeding parts assume that you have basic knowledge of programming in C. We will present the C language in general context then we'll introduce it within the contest of use in PIC programming. The following discussion attempts to write and simulate a simple C program in MPLAB and check the results

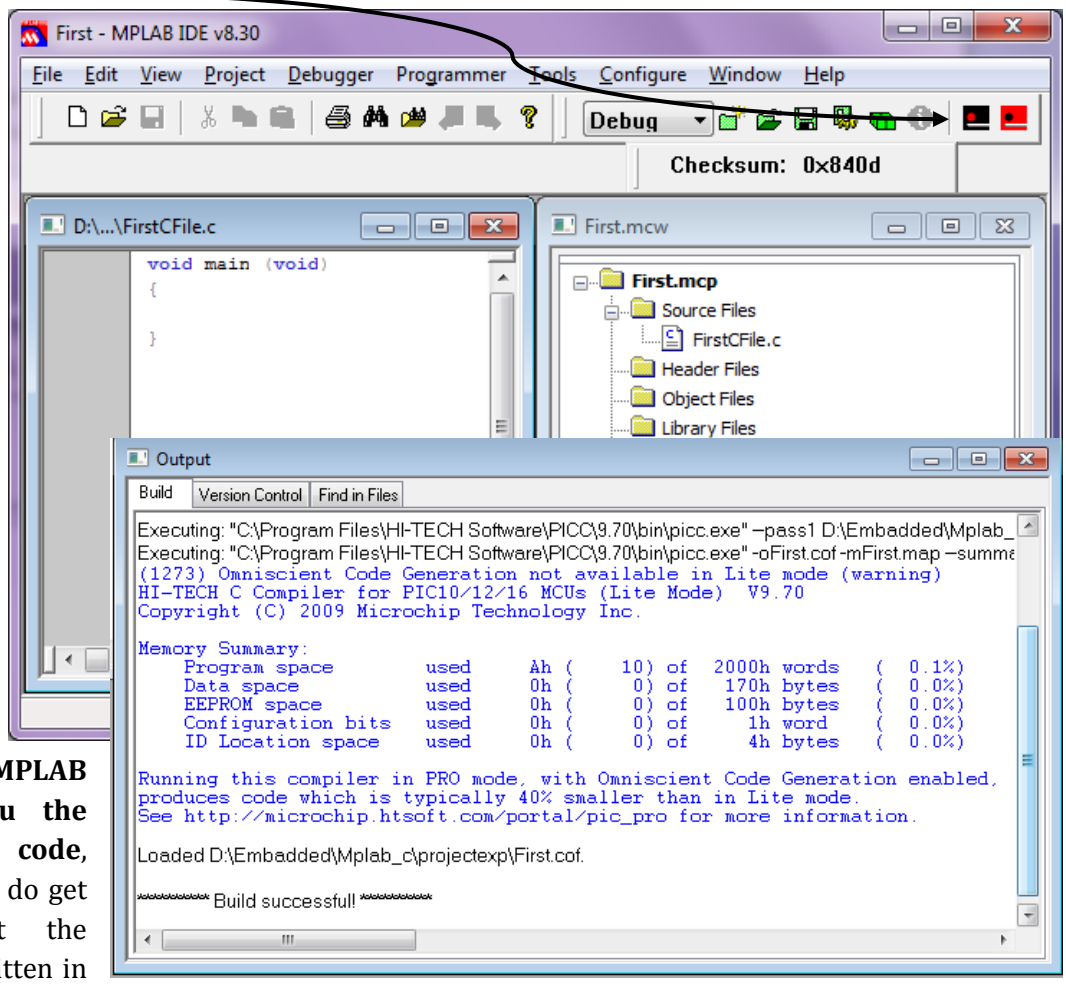
In MPLAB, inside your newly created project from above, write the following:

```
#include <htc.h>
void main(void)    // every C program you write needs a function called main.
{

}
```

Notice that comments are indicated with // instead of ';'

After writing the above EMPTY program we should build the code to ensure that MPLAB IDE and HI-TECH C are properly installed. Select Build from the Project menu, or choose any of MPLAB IDE's shortcuts to build the project — you can, for instance, click on the toolbar button that shows the HI-TECH "ball and stick" logo, as shown in the figure below. You will notice that the Project menu has two items: Build and Rebuild.



An output window should show with BUILD SUCCEDDED

The compiler has produced memory summary and there is no message indicating that the build failed, so we have successfully compiled the project. If there are errors they will be printed in Build tab of this window. **You can double-click each error message and MPLAB IDE will show you the offending line of code**, where possible. If you do get errors, check that the program is as it is written in this document. BUILD SUCCEED DOES NOT MEAN THAT YOUR PROGRAM IS CORRECT, IT SIMPLY MEANS THAT THERE ARE NO **SYNTAX** ERRORS FOUND, SO WATCH OUT FOR ANY LOGICAL ERRORS YOU MIGHT MAKE.

Quick Review of the Basic Rules for Programming in C

1. Comments for only ONE line of code start with 2 slashes: //
   // This is a one line comment
   Remember to always document your code through the use of functional comments!
2. Comments for more than one line start with /* en end with */
   /*
   This is a comment.
   This is another comment.
   */
3. At the end of each line with some instruction, a semi-colon (;) has to be placed.
   a=a+3;
4. Parts of the program that belong together (functions, statements, etc.), are between { and }.
   void main(void) //Function
   {
   //Add code
   }

The Basic Structure of a C Program

The ***ordered*** structure of a program in C is as follows:

- Libraries
- Global Variables
- Function Prototypes
- Main Function
- Functions

❖ Adding libraries (the initial few lines of any C program)

Syntax: #include <filename.h>

Libraries such as "htc.h", "math.h" and "stdlib.h" include many references to built-in variables and functions to be used in programs, if the header files are not included, the built-in functions and variables if used will not be defined which will result in build errors.

The htc.h file will be included in all our C programs which use the HI-TECH compiler, other compilers have different header files, refer to their documentation when needed.

❖ Declaring "global" variables.

Define and declare the variables to be used throughout the program, this is in contrast to "local" variables discussed later on.

❖ Defining prototypes of the functions.

A C program has a main function and possibly other functions as well which might be written below the main function. If we are to call any of the other functions from inside the main subroutine, the build will fail and indicate that the function is undefined. This is because the code is compiled line by line and at the moment the compiler attempts to compile "call function", it still has not known of the existence of this function because it is declared later in the code "after main". One solution is to *place all the functions before the main function*. Another preferred method is the use of function *prototype*. A prototype of a function ensures that the function can be called anywhere in the program**. It is simply copying only the header of the function, placing it before the main subroutine and ending it with a semicolon ';'**

- Main function.

  This is the function that will be called first when starting your microcontroller. From there, other functions are called. ***Every C program must have a main function.***

- Functions.

  Functions are a grouping of instructions which perform a certain task. They are the unit of modularity and are very useful to make it easy to repeat tasks. They have input and output variables.

Syntax: type identifier function name (type identifier identifier1, type identifier identifier2 ….)

```
{
        //The body of the function
        return identifier        //only when return type is not void
}
```

Type identifier: could be int, long, short, char, void ….. etc

The output variable type precedes the function's name, input variables follow the function name and are placed in between brackets, a function can take as many input variables as needed but it only returns one output variable.

testFunction1 has two input parametres of type integer (x,y) but has no output, all processing is local inside the function and it returns no values	testFunction2 has one input parameter of type integer (x), it returns an output which is the square of the input number. Notice, that value returning functions end with a return statement, omitting of which will result in an error	testFunction3 takes no input or outputs.

`void testFunction1(int x, int y)`
`{`
`  int k;`
`  k = x;`
`  y = 2 + x;`
`}`
	`int testFunction2 (int x)`
	`{`
	`  return x*x;`
	`}`
		`void testFunction3 (void)`
		`{`
		`  //some code`
		`}`

How to call function: Examples		
testFunction1(75,99)	A = testFunction2(5) Since this type of functions returns a value, the value need be stored in a previously defined variable. The variable must be defined as the same return output type of the function, if the function returns an integer, A must be defined as integer, if the function returns a character, A need be defined as character …	testFunction() Note that the brackets are left empty when no arguments are passed

Example Program 1: Typical Program Layout

```
// ExampleProgram1.c
        #include <htc.h> //Always include this library when using HI-TECH C compiler
//Declaring global variables
        int     a, b, c;
        char    temp;
//Defining prototypes
        int     calc (int p);
//Main function
        void main(void)
        {
                a=calc(3); //write main body code
        }
//Functions
        int calc (int p)
        {
                p=p+1;     //write function body code
                return p;
        }
```

More on Variables

Variables can be classified into two main types depending on their scope:

Global Variables

These variables can be accessed (i.e. known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled. *In Example Program 1, (a, b, c, and temp) are **GLOBAL VARIABLES***

Local Variables

These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called. *In Example Program 1, (p) is a **LOCAL VARIABLE***

Variable Types

The following table lists all possible variable types in C, the size they take up in memory and the range of each.

Type	Memory usage	Possible values
bit	1 bit	0, 1
char	8 bits	-128…127
unsigned char	8 bits	0…255
signed char	8 bits	-128…127
int	16 bits	-32k7…32k7
unsigned int	16 bits	0…65k5
signed int	16 bits	-32k7…32k7
long int	32 bits	-2G1…2G1
unsigned long int	32 bits	0…4G3
signed long int	32 bits	-2G1…2G1
float	32 bits	± 10^(±38)
double	32 bits	± 10^(±38)

Default Input Is Decimal

Example Program 2:

```
#include <htc.h>
char            Ch;
unsigned int    X;
signed int      Y;
int             Z, a, b, c; // Same as "signed int"
unsigned char   Ch1;
bit             S, T;

void main (void)
{
        Ch = 'a';
        X  = -5;
        Y  = 0x25;
        Z  =-5;
        Ch1='b';
        T   = 0;
        S   = 81;       //S=1 When assigning a larger integral type to a bit variable,
                        //only the Least Significant bit is used.
        a   = 15;
        b   = 0b00001111;
        c   = 0x0F;
        // a, b, c will all have the same value which is 15

}
```

C Operators

- Relational and bit operators

>	Greater than
>=	Greater than or similar to
<	Less than
<=	Less than or similar to
==	Equal to
!=	Not equal to

~	Bitwise NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Shift to left
>>	Shift to right

- Arithmetic operators

x--;	This is the same as x = x – 1;
x++;	This is the same as x = x + 1;

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder after division)

Operators Precedence Chart

Operator precedence describes the order in which C reads expressions. For example, the expression a=4+b*2 contains two operations, an addition and a multiplication. Does the HI TECH compiler evaluate 4+b first, then multiply the result by 2, or does it evaluate b*2 first, then add 4 to the result? The operator precedence chart contains the answers. Operators higher in the chart have a higher precedence, meaning that the HI TECH compiler evaluates them first. Operators on the same line in the chart have the same precedence, and the "Associativity" column on the right gives their evaluation order.

Operator Precedence Chart		
Operator Type	**Operator**	**Associativity**
Primary Expression Operators	**()**	left-to-right
Binary Operators	*** / %**	left-to-right
	+ -	
	>> <<	
	< > <= >=	
	== !=	
	&	
	^	
	|	

Example Program 3: Fibonacci series: 0, 1, 1, 2, 3, 5

```
#include <htc.h>                                          // Library
unsigned int Fib (unsigned int Num1, unsigned int Num2); // Prototype
unsigned int    F1, F2, F3, F4, F5, F6;                   // Global Variables

void main (void)                                          // Main function
{
        F1 = 0;
        F2 = 1;
        F3 = Fib (F1, F2);
        F4 = Fib (F2, F3);
        F5 = Fib (F3, F4);
        F6 = Fib (F4, F5);
}
 unsigned int Fib (unsigned int Num1, unsigned int Num2) //Function
{
        return Num1 + Num2;
}
```

Preparing for Simulation

1. Start a new MPLAB session, add the file *ExampleProgram3.c* to your project
2. Build the project
3. Select **Debugger** ↳ **Select Tool** ↳**MPLAB SIM**
4. Go to View Menu → Watch (From the drop out menu choose the variables watch F1 through F6 we want to inspect during simulation and click ADD Symbol for each one)

From the **Debugger Menu** → choose **Select Tool** → then **MPLAB SIM**
After the following buttons appears in the toolbar:

5. Press the "Step into" button one at a time and check the Watch window each time an instruction executes; keep pressing "Step into" until you all the six terms of the series are generated.
6. Reset the simulation, do step 5 above but this time use "Step Over", note the difference
7. Reset the simulation, do step 5 above, this time place a break point at the last instruction in main, press run. Inspect the variables in watch window.

Notes about simulating a code written in C in MPLAB

Stepping into codes written in C is not as direct as one would imagine, different compilers translate the C code into assembly differently, a single line of code might be translated into multiple assembly lines, for example a simple assignment statement "X = 5" where X has been defined as integer will be translated into four assembly instructions.

```
Movlw 05
Movwf 0x70   //GPR address 0x70 chosen by compiler
Movlw 00
Movwf 0x71
```

Since X is an integer which reserves 2 bytes in memory (16 bits as specified in the table in page 7), it need be saved as 0x0005, so two instructions are needed to load the first byte into location 0x70 and another two to move the rest of the number into location 0x71.

If a simple one statement instruction was assembled like this, imagine how would complex statements be translated like for loops and if statements. Moreover, some compilers are more efficient than others, which give you optimized shorter assembly codes which might not be easy to understand.

Moreover, function placement spans through multiple pages in program memory, the code might not be placed in consecutive order into memory by the compiler; further overhead instructions to switch between pages are common.

In addition, the use of built-in library functions will further complicate stepping through assembly codes line by line as these functions are often provided as a black box for the developer to use with no interest in their details.

For this, it might be difficult for the inexperienced to understand the assembly code generated by compilers, and stepping into assembly code one instruction at a time might be a headache. ***It is often advised to place breakpoints at points of interest and run the program till it halts at the required breakpoints and analyze the outputs in the watch window.***

Control and Repetition Statements

- IF...ELSE statements

```
if (expression1)
{
  statement 1;
        .
        .
  statement n;
}
else
{
  statement 1;
        .
        .
  statement n;
}
```

Example Code 5:

```
if (a==0) //If a is equal to 0
{
        b++; // increase b and c by 1
        c++;
}
else
{
        b--; //decrease b and c by 1
        c--;
}
```

- WHILE loop

```
while (expression)
{
  statement 1;
  statement 2;
        .
        .
  statement n;
}
```

Example Code 6:

```
while (a>=1 ) && (a <=10) //As long as 1<=a <= 10
{
        b = b + 3;
        c = a%b;
}
```

- FOR loop

```
for (expr1; expr2; expr3)
{
  statement 1;
  statement 2;
        .
        .
  statement n;
}
```

Example Code 7:

```
for (i = 0 ; i < 100 ; i++) //loop 100 times
{
        B = B + i + A%i;
}
```

C for PIC

The preceding discussion introduced the C language in a broad concept. Now, we will draw an example of how to use C with the PIC microcontroller. Actually, it is fairly simple where besides user defined variables, the PIC registers are also used in the context of programs.

The microcontroller is completely controlled by registers. All registers used in MPLAB HI-TECH have exact the same name as the name stated in the datasheet. Registers can be set in different ways, following are few examples:

```
TRISB   = 0b00000000;  //TRISB is output
PORTC   = 255;         //All pins of PORTC are made high
PORTD   = 0xFF;        //All pins of PORTD are made high
PORTB   = 170;         //Pin B7 on, B6 off, B5 on, B4 off, etc.
TRISB   = 0b11110010;  //Pin RB7, RB6, RB5, RB4 and RB1 are input, other bits are outputs.
OPTION=0xD4            //PSA assigned to TMR0, Prescalar = 32, TMR0 clock source is the internal instruction cycle
                       //clock, External interrupt is on the rising "refer to datasheet"
```

To set or reset one single bit in a register (one of the 8 bits), the pin name is used and, the names of the bits are also as specified and used in the datasheet.
Some examples:

```
RB0 = 1 //Pin B0 on
RB7 = 0 //Pin B7 off
```

Example Program 8: Periodically switch a LED connected to RD0 on and off

```
#include <htc.h>
// if the whole function is placed before the main function, there is no need for a prototype
void Wait()
{
        unsigned char i;
        for(i=0; i<100; i++)
        _delay(60000);          //built in function .. more info next page
}

void main()
{
  //Initialize PORTD -> RD0 as Output
        TRISD=0b11111110;

  //Now loop forever blinking the LED.
        while(1)
        {
            RD0 = 1;    //LED on
            Wait();

            RD0 = 0;    //LED off
            Wait();
        }
}
```

To simulate the above example code, you can either select PORTD from the ADD SFR drop down menu or choose _PORTDbits from the ADD SYMBOL drop list, click on the + sign to expand and see the individual bits.

Place your break points on both Wait() instructions and run the code.

BUILT IN LIBRARY FUNCTIONS

The C standard libraries contain a standard collection of functions, such as string, math and input/output routines. The declaration or definition for a function is found in the htc.h and other libraries files which are to be included whenever necessary. Some of these functions are listed below, the syntax of each and a brief description follows.

Delay functions

_DELAY

Synopsis

```
#include <htc.h>
void _delay(unsigned long cycles);
```

Description

This is an inline function that is expanded by the code generator. The sequence will consist of code that delays for the number of cycles that is specified as argument. The argument must be a literal constant.
An error will result if the delay period requested is too large. **For very large delays, call this function multiple times.**

//Example

```
#include <htc.h>
int A;


void main (void)
{
        A = A | 0x7f;
        _delay(10); // delay for 10 cycles
        A = A & 0x85;
 }
```

__DELAY_MS, __DELAY_US

Synopsis

```
__delay_ms(x) // request a delay in milliseconds
__delay_us(x)  // request a delay in microseconds
```

Description

As it is often more convenient request a delay in time-based terms rather than in cycle counts, the macros __delay_ms(x) and __delay_us(x) are provided. These macros simply wrap around _delay(n) and convert the time based request into instruction cycles based on the system frequency.
These macros require the prior definition of preprocessor symbol _XTAL_FREQ. This symbol should be defined as the oscillator frequency (in Hertz) used by the system.

//Example

```
#include <htc.h>
int A;
#define _XTAL_FREQ  4000000

void main (void)
{
        A = A | 0x7f;
         __delay_ms(10); // delay for 10 ms
        A = A & 0x85;
}
```

Arithmetic functions

In addition to the htc.c library, other libraries such as Standard Library <stdlib.h> or C Math Library <math.h> need be included in the project for making use of many useful built-in functions. Make sure you include the appropriate header files for each library before making use of its functions or else build errors will be present.

ABS, POW, LOG, LOG10, RAND, MOD, DIV, CEIL, FLOOR, NOP, ROUND, SQRT are required.. refer to the datasheet for the documentation of the others

ABS

Synopsis

```
#include <stdlib.h>
int abs (int j)
```

Description

The abs() function returns the absolute value of the passed argument j.

POW

Synopsis

```
#include <math.h>
double pow (double f, double p)
```

Description

The pow() function raises its first argument, f, to the power p.

LOG, LOG10

Synopsis

```
#include <math.h>
double log (double f)
double log10 (double f)
```

Description

The log() function returns the natural logarithm of f. The function log10() returns the logarithm to base 10 of f.

RAND

Synopsis

```
#include <stdlib.h>
int rand (void)
```

Description

The rand() function is a pseudo-random number generator. It returns an integer in the range 0 to 32767, which changes in a pseudo-random fashion on each call.

Trigonometric functions

SIN, COS, TAN, COS, ASIN, ATAN …… refer to the data sheet for the others

➢ **SIN**	➢ **COS**
Synopsis #include <math.h> double sin (double f) ***Description*** This function returns the sine function of its argument.it is very important to realize that C uses radians, not degrees to perform these calculations! If the angle is in degrees you must first convert it to radians.	***Synopsis*** #include <math.h> double cos (double f) ***Description*** This function yields the cosine of its argument, which is an angle in radians. The cosine is calculated by expansion of a polynomial series approximation.

```
// Example:
#include <htc.h>
#include <math.h>
#include <stdio.h>
#define C 3.141592/180.0
double X,Y;
void main (void)
{
double i;
X=0;
Y=0;
for(i = 0 ; i <= 180.0 ; i += 10)
{X= sin(i*C);
Y= cos(i*C);
}
}
```

➢ define directive

You can use the **#define** directive to give a meaningful name to a constant in your program.

```
#define identifier constant
```

Example:
#define COUNT 1000