BoostBasic Compiler

for PICmicro

Reference Manual

V1.12

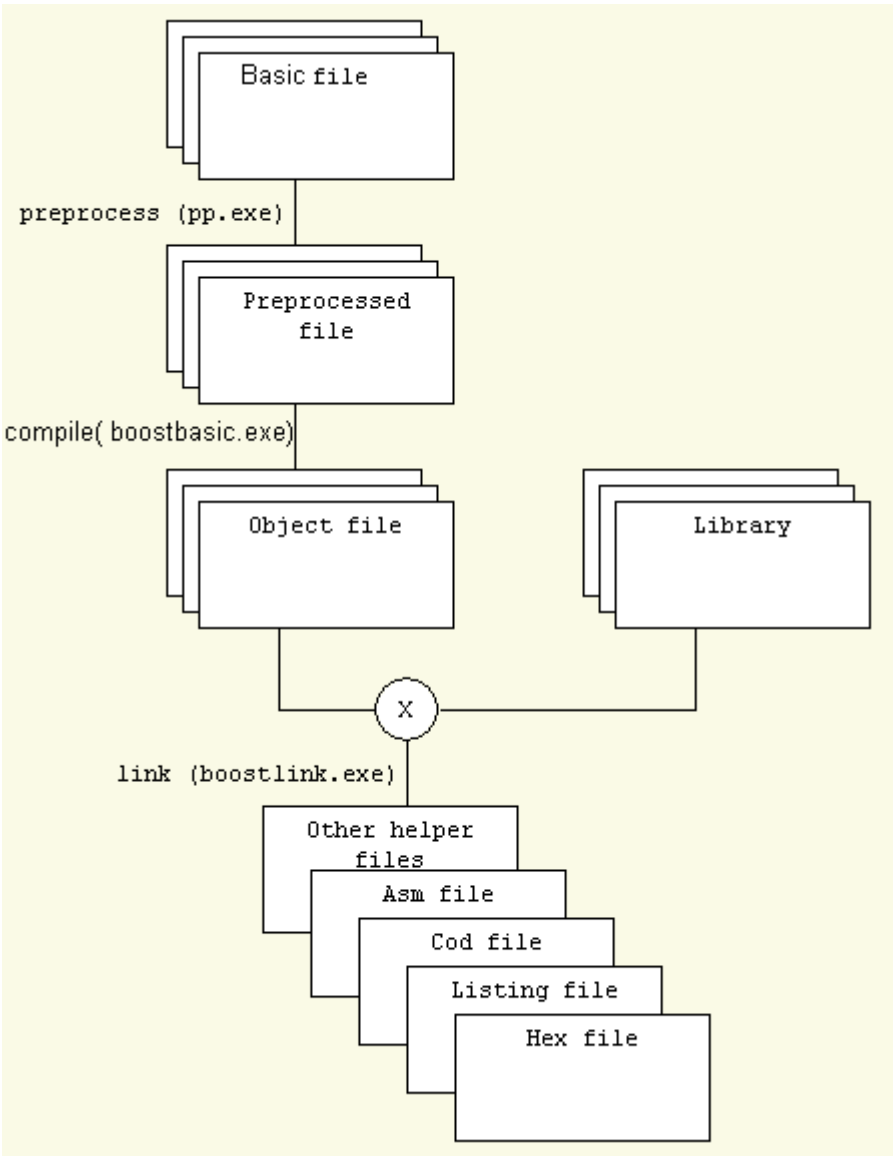
Index

This page is intentionally left blank.

Introduction

BoostBasic™ is a Basic compiler that works with PIC16, PIC18 and some PIC12 processors.

Compilation model



Compiler

There are two separate compilers for pic16 and pic18 targets. There is no need to specify which one to use if you work under SourceBoost™ IDE. It picks compiler based on the selected target. The output of the compiler is one or more .obj files that should be processed further by librarian or linker to get .lib or .hex file.

Librarian

Librarian is built into BoostLink™ linker executable and gets activated by `-lib` command line argument. There is a dedicated box in the Option dialog inside SourceBoost™ IDE that makes project generate a library instead of hex file.

Linker

BoostLink™ Optimizing Linker links .obj files generated by compiler into .hex file that is ready to send to target. It also generates some auxiliary files used for debugging and code analysis.

MPLAB integration

BoostBasic compiler can be integrated into Microchips MPLAB integrated development environment (IDE). The MPLAB integration option should be selected during the SourceBoost software package installation.

Please note: To use BoostBasic under MPLAB the MPLAB integration button must be pressed during the SourceBoost package installation. This copies some files and sets the required registry keys required for integration to work.

In case the installation step "MPLAB Integration" failed, the files in the <SourceBoost>\mplab directory can be manually copied into

<MPLAB IDE>\MPLAB IDE\Core\MTC Suites for **MPLAB 8.x,** or

<MPLAB IDE>\Third Party\MTC Suites for **MPLAB 7.x,** or

<MPLAB IDE>\LegacyLanguageSuites for **MPLAB 6.x**.

In the above examples, <MPLAB IDE> refers to the MPLAB installation directory and <SourceBoost> refers to the SourceBoost IDE and compilers installation directory.

Features

When **BoostBasic** is integrated into MPLAB IDE it allows the following:

- Use of the MPLAB Project Manager within MPLAB IDE.
- Creation and Editing of source code from within MPLAB IDE.
- Build a project without leaving MPLAB IDE.
- Source level debugging and variable monitoring using: MPLAB simulator, MPLAB ICD2, MPLAB ICE2000.

Setting the MPLAB Language Tool Locations

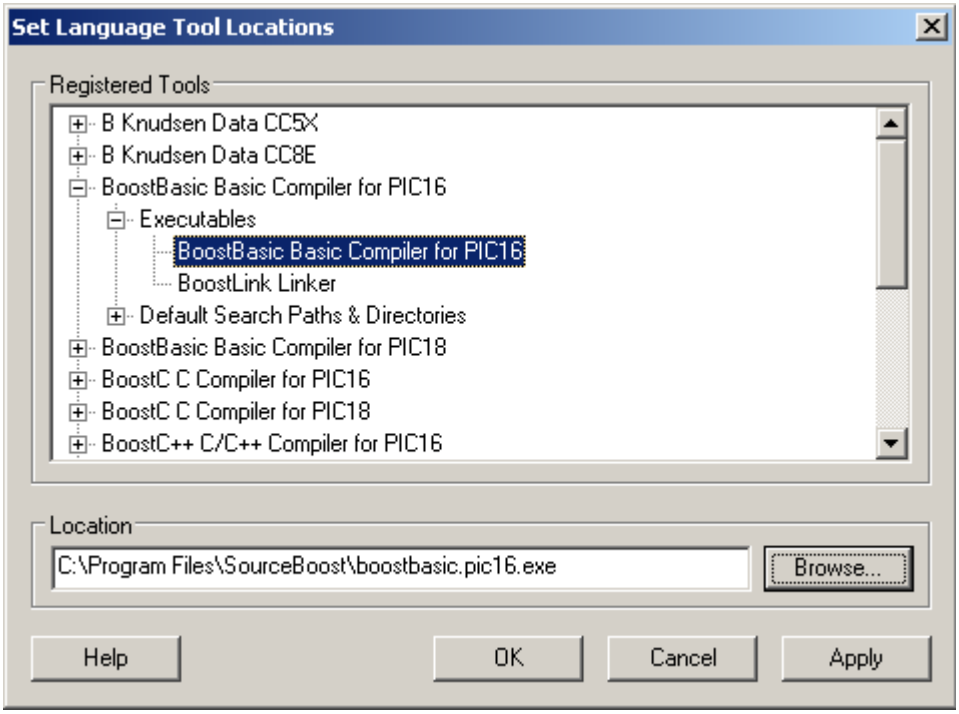
Note: this process only needs to be performed once.

The procedure below specifies paths assuming the default installation folder has been used for the SourceBoost software package.

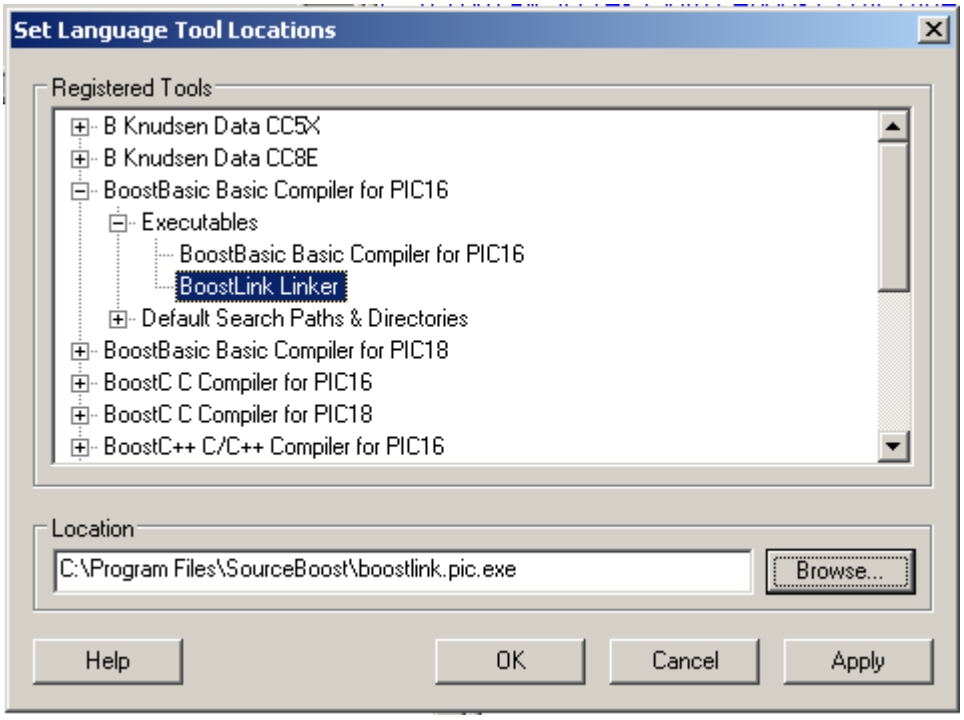
1. Start MPLAB IDE.

2. Menu **Project ➡ Set Language Tool Locations**.
   Note: if BoostBasic compiler does not appear in the Registered Tools list, then the integration process during the SourceBoost installation was not performed or was unsuccessful.

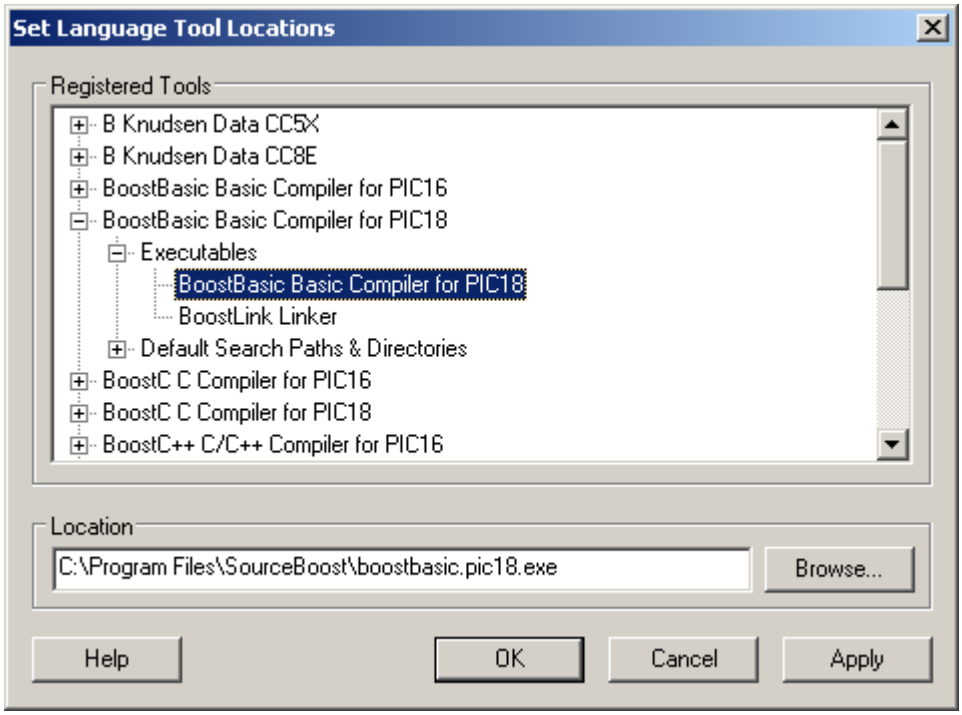
3. Set **BoostBasic** compiler for PIC16 location:



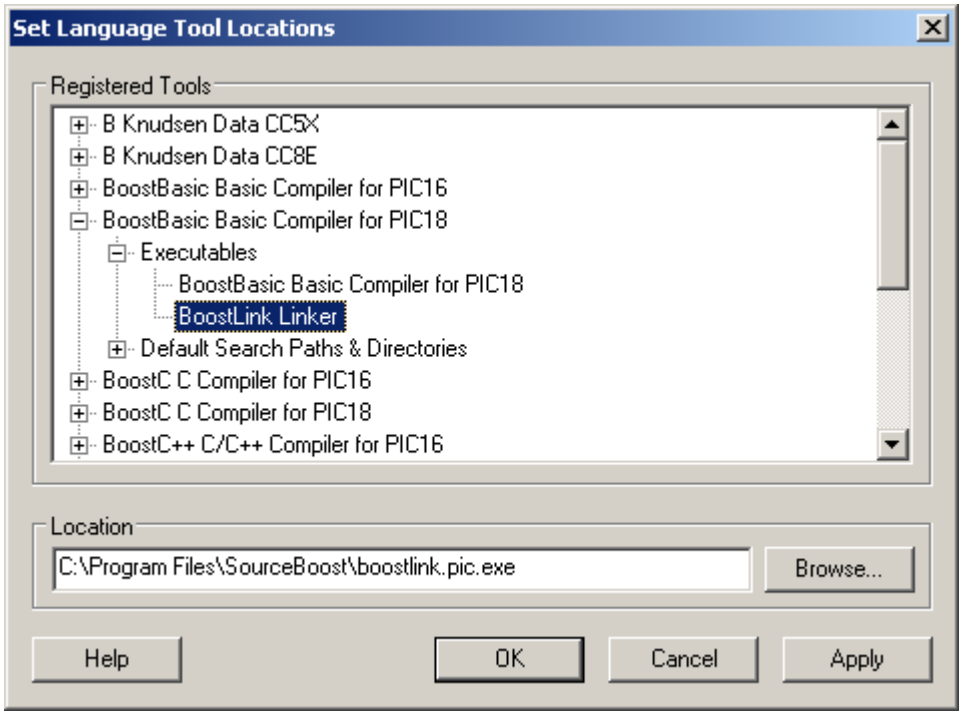
4. Now set **BoostLink** Linker location:



5. Set **BoostBasic** compiler for PIC18 location:



6. Eventually, set **BoostLink** Linker location in the PIC18 tree:



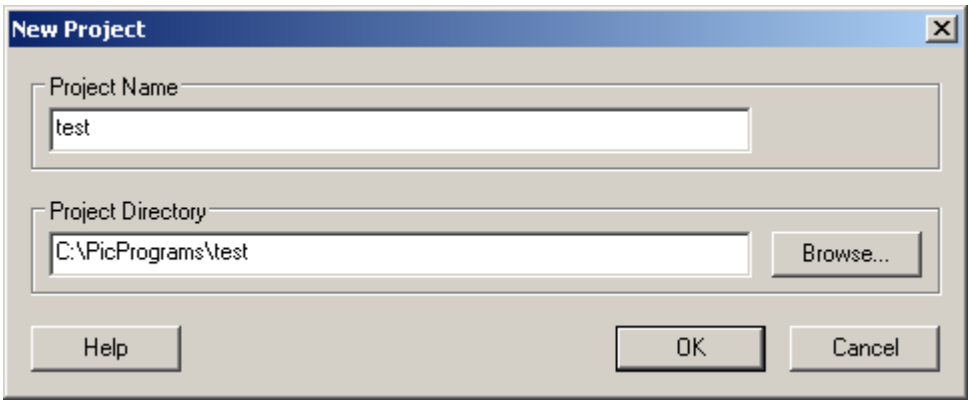
Creating a project under MPLAB IDE

Before attempting to do this, please ensure that the "**Setting the MPLAB language tool locations"** process illustrated in the above section has been successfully performed.

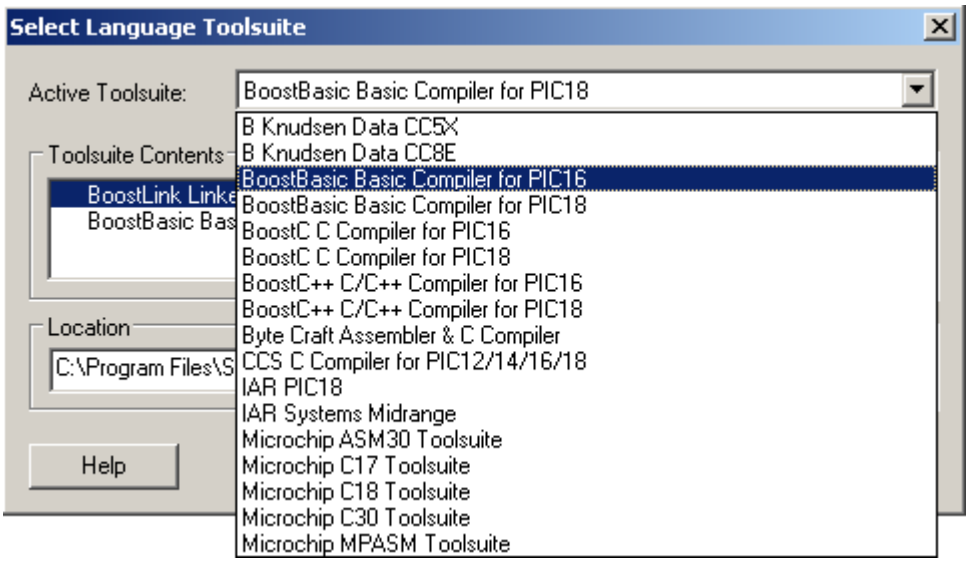
The following steps will help you create a project under MPLAB IDE, that will be built using the BoostBasic compiler, compiling for a PIC16 Target. The project

name is test and the project and source code will be located in folder C:\PicPrograms\test

1. Menu **Project ➡ New**. Enter a project name and directory.
   Note: this can be an existing directory containing a SourceBoost IDE project.

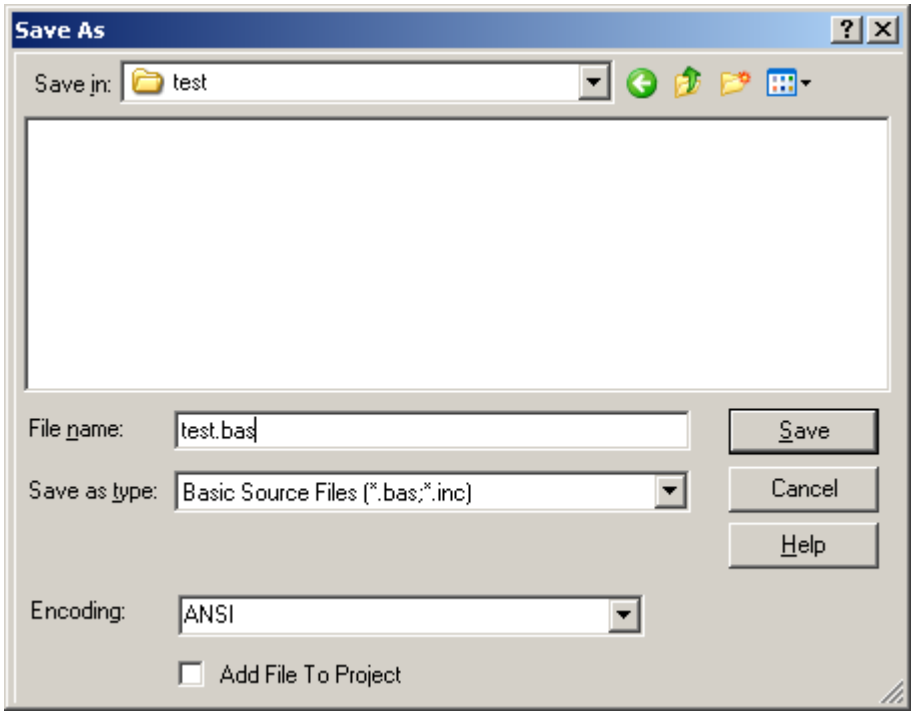


2. Menu **Project ➡ Select Language Toolsuite**. Select the **BoostBasic** Compiler for PIC16.

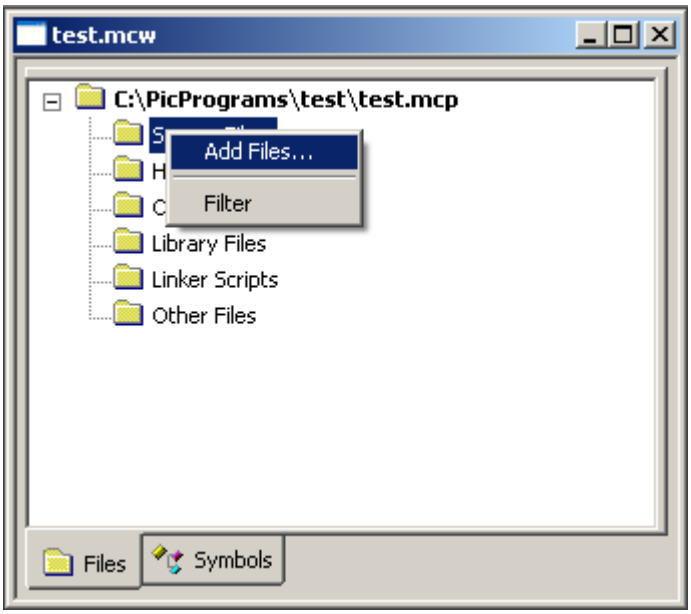


3. Menu **File ➡ New**. Type code into the Untitled window.
   Note: If you already have Source Files, steps 2 and 3 can be skipped.

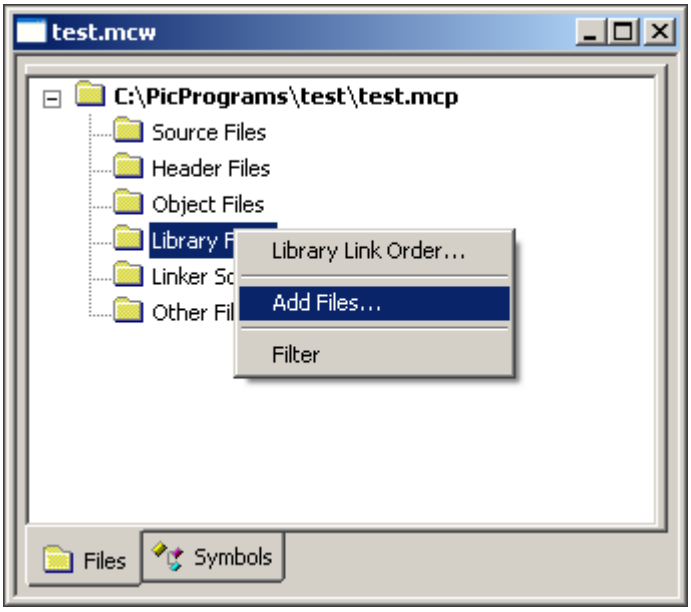
4. Menu **File ➡ Save As**. Locate the project folder using the Save As dialog box.



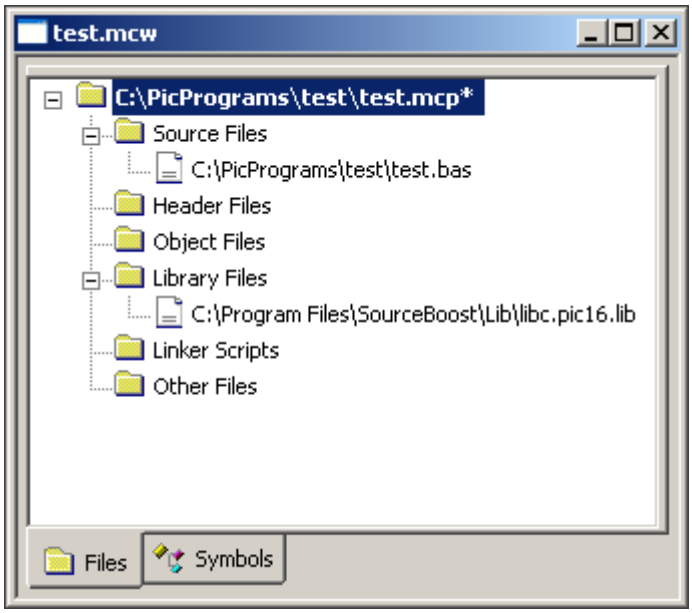
5. Add the test.c source file to the project by right clicking on Source Files in the project tree – as shown below.



6. Add the libc.pic16.lib file (found in the C:\Program Files\SourceBoost\Lib folder) to the project by right clicking on Library Files in the project tree.



7. Check the final project. It should look as below:



8. Menu **Project ➡ Build** (or press the build button on the tool bar). The code should then be built.

You can now use the MPLAB simulator, ICD2 or ICE to run the code, or a programmer to program a device. Please refer to the "Using ICD2" section of this document before using ICD2 to avoid potential problems.

Limitations under MPLAB

Currently we recommend only creating projects with a single .bas source file in them. The source file should start have #include <basic\system.bas> before any code.

Example .bas file that compiles under MPLAB:

```
#include <basic\system.bas>

Rem PIC18F8720 configuration
#pragma DATA    _CONFIG1H, _OSCS_OFF_1H & _HS_OSC_1H
#pragma DATA    _CONFIG2L, _BOR_ON_2L & _BORV_20_2L & _PWRT_OFF_2L
#pragma DATA    _CONFIG2H, _WDT_OFF_2H & _WDTPS_128_2H
#pragma DATA    _CONFIG3H, _CCP2MX_ON_3H
#pragma DATA    _CONFIG4L, _STVR_ON_4L & _LVP_OFF_4L & _DEBUG_OFF_4L
#pragma DATA    _CONFIG5L, _CP0_OFF_5L & _CP1_OFF_5L & _CP2_OFF_5L &
_CP3_OFF_5L
#pragma DATA    _CONFIG5H, _CPB_OFF_5H & _CPD_OFF_5H
#pragma DATA    _CONFIG6L, _WRT0_OFF_6L & _WRT1_OFF_6L & _WRT2_OFF_6L &
_WRT3_OFF_6L
#pragma DATA    _CONFIG6H, _WRTC_OFF_6H & _WRTB_OFF_6H & _WRTD_OFF_6H
#pragma DATA    _CONFIG7L, _EBTR0_OFF_7L & _EBTR1_OFF_7L & _EBTR2_OFF_7L &
_EBTR3_OFF_7L
#pragma DATA    _CONFIG7H, _EBTRB_OFF_7H

#pragma CLOCK_FREQ 20000000

Sub PortUpdate()

     latd = latd + 1

End Sub

Sub main()

     trisd = 0 ' set port to output
     latd = 0 ' initialise port data

     Do While 1

          Rem increase port count value every 250ms
          Call PortUpdate()
          Call delay_ms( 250 )

     Loop


End Sub
```

Using ICD2

The are a few things to be aware of when using or planning using ICD2:

1. **RAM usage**: ICD2 uses some of the target devices RAM, leaving less room for the actual application.

   In order to reserve the RAM required by ICD2, and prevent Boost Linker from using it, the *icd2.bas* file must be included in the source code, eg:

```
#include <basic\system.bas>
#include <basic\icd2.bas>       ' allocates RAM used by ICD2

Sub main()

    Do While 1
    Loop

End Sub
```

2. **SFR usage**: ICD2 uses some Special Function Registers. This prevents the use of some peripheral devices when using ICD2 to debug code.

   Important: It is down the user to ensure that the ICD2 special function registers are not accessed. On some targets these registers reside at the same address as other peripheral device special function registers. Please check the documentation provided in the MPLAB IDE help for ICD2 resource usage in order to prevent problems.
3. **Break point overrun**: Due to timing skew in the target device (caused by instruction prefetch), execution will pass the instruction address where a breakpoint is set before it stops.
4. **NOP at ROM address 0**: See the BoostLink command line option -icd2 to add a NOP at ROM address 0.

Command line options

To get full list of BoostBasic compiler and BoostLink linker command line options run compiler or linker from command line.

BoostBasic command line

```
BoostBasic Optimizing Basic Compiler Version x.xx

http://www.sourceboost.com
Copyright(C) 2004-2009 Pavel Baranov
Copyright(C) 2004-2009 David Hobday

Licensed to <license info>

Usage: boostbasic.pic16.exe [options] files

Options:
        -t name         target processor (default name=PIC16F648A)
        -On             optimization level (default n=2)
                        n=0 - optimization turned off
                        n=1 - optimization turned on
                        n=2 - global optimization turned on
                        n=a - aggressive optimization turned on
                        n=p - 32 bit long promotion turned on
        -Wn             warning level (default n=1)
                        n=0 - no warnings
                        n=1 - some warnings
                        n=2 - all warnings
        -Werr           treat warnings as errors (default off)
        -i              debug inline code (default off)
        -Su             disable initialization of uninitialized static variables
        -d name         define 'name'
        -m              generate dependencies file (default off)
        -v              verbose mode turned on (default off)
        -I path1;path2  additional include directories
        -beep           issue sound at the end of compilation (default off)
```

Optimization

Code optimization is controlled by -O command line option and *#pragma*.

Optimize flags:

-O0 no or very minimal optimization

-O1 regular optimization

-O2 global optimization (this option is recommended for most applications)

-Oa aggressive optimization (produces shorter code and optimizes out some variables - this can make debugging more difficult!)

-Op promotes results of some 16 bit operations to 32 bits (can result in more efficient code is some cases).

BoostLink command line

```
BoostLink Optimizing Linker Version x.xx
http://www.sourceboost.com
Copyright(C) 2004-2009 Pavel Baranov
Copyright(C) 2004-2009 David Hobday

Licensed to <license info>

Usage: boostlink.pic.exe [options] files
Options:
  -t name      target processor
  -On          optimization level 0-1 (default n=1)
               n=0 - no optimization
               n=1 - pattern matching and bank switching optimize on
  -v           verbose mode
  -d path      directory for project output
  -p name      project (output) name for multiple .obj file linking
  -ld path     directory for library search
  -rb address  ROM base (start) address to use
  -rt address  ROM top (end) address to use
 -swcs s1 s2 s3 Use software call stack. Hardware stack is allocated by
                specifying stack depths s1,s2,s3 (optional)
                s1 = main and task routines hardware stack allocation
                s2 = ISR hardware stack allocation
                s3 = PIC18 low priority ISR hardware stack allocation
 -isrnoshadow   ISR No use of Shadow registers
 -isrnocontext  ISR No context Save/restore is added to ISR(PIC18 only)
 -icd2          Add NOP at first ROM address for correct ICD2 operation
 -hexela        Always add extended linear address record to .hex file
 -beep          Issue sound at the end of link (default off)

Switches for making libraries:
  -lib         make library file from supplied .obj and .lib files
  -p name      project (library output file) name
```

-rb

This command line option causes the code generated by the linker to start at the address specified. Boot loaders often reside in the low area of ROM.

Example

-rb 0x0800

-swcs s1 s2 s3

This command line option to the linker tells it to use a software call stack in addition to the hardware call stack. This allows subroutine calls deeper than the call hardware call stack of the PIC. A function call that is made on the software call stack uses an extra byte of RAM to hold the return point number. Where possible the hardware stack is used for efficiency. By specifying the depth of hardware stack to use for main (and Novo tasks) ***s1***, ISR (interrupt service routine) ***s2*** and low priority ISR (PIC18 only) ***s3***, provides control over when the software call stack is used instead of the hardware call stack. The software call stack is applied to functions higher up in the call tree, so calls lower down the call tree still use the hardware call stack.

Example:

-swcs 6 2

Main routine will use hardware call stack up to a depth of 6 and then start using software call stack. Interrupt routine will use hardware call stack up to a depth of 2 and start using software call stack. An ISR uses hardware call stack depth of 1 to save the address of the point where the code was interrupted, so in this example it only leaves a hardware call stack depth 1 for subsequent calls within the ISR.

-isrnoshadow

This command line switch tells the linker not to use the PIC18 shadow registers for interrupt service routine (ISR) context saving. This option is required as a work around for silicon bugs in some PIC18's.

-isrnocontext

This option only works with PIC18's. When use this prevents the linker adding extra code for context saving. This allow the programmer to generate their own minimal ISR context saving code, or have none at all.

Example:

```
// Context saving example
// Assumes that the ISR code will only modify w and and bsr

// create context saving buffer at fixed address
char context[ 2 ]@0x0000;

void interrupt()
{
     asm movff _bsr, _context
     asm movwf _context+1
     ....
     asm movwf _context+1
     asm movff _bsr, _context
}
```

-icd2

Use this command line switch to add a NOP instruction at the first ROM address used (usually address 0). This is required on some devices for correct operation of Microchip ICD2 (In Circuit Debugger).

-hexela

Always add extended linear address record to .hex file. Without this switch an extended linear address record is only added to the .hex file if required by addresses included in the .hex file.

libc Library

When a project is being linked, **SourceBoost IDE** adds *libc.pic16.lib* or *libc.pic18.lib* to the linker command line, if it can find this library in its default location.
The *libc* library contains necessary code for multiplication, division and dynamic memory allocation. It also includes code for string operations.

Code entry points

Entry points depend on the code address range using by the BoostLink linker. By default, the linker uses all available code space, but it's also possible to specify code start and end addresses that linker should use through linker command line options.

For PIC16:

Reset (main) entry point	<code start> + 0x00
Interrupt entry point	<code start> + 0x04

For PIC18:

Reset (main) entry point	<code start> + 0x00
Interrupt entry point	<code start> + 0x08
Low priority ISR entry point	<code start> + 0x18

Pragma directives

Specific BoostBasic preprocessor directives all follow the keyword *#pragma.*

The following directives are supported by **pp**:

#pragma DATA

#pragma CLOCK_FREQ

#pragma OPTIMIZE

These directives are individually explained in the following pages.

#pragma DATA

Syntax: *#pragma DATA* **addr, d1, d2, ...**

or

#pragma DATA **addr, "abcdefg1", "abcdefg2", ...**

Elements: **addr** is any valid code memory address.

d1, d2... are 8-bit integer constants.

"abcdefgX" is a character string, the ASCII values of the charcters will be stored as 8 bit value.

Purpose: User data can be placed at a specific location using this construct. In particular, this can be used to specify target configuration word or to set some calibration/configuration data into on-chip eeprom.

Examples:

```
#pragma DATA 0x200, 0xA, 0xB, "test"
Rem Set PIC16 configuration word
#pragma DATA 0x2007, _HS_OSC & _WDT_OFF & _LVP_OFF
Rem Put some data into eeprom
#pragma DATA 0x2100, 0x12, 0x34, 0x56, 0x78, "ABCD"
```

#pragma CLOCK_FREQ

Syntax: *#pragma CLOCK_FREQ* `Frequency_in_Hz`

Elements: **Frequency_in_Hz** is the processor's clock speed.

Purpose: The CLOCK_FREQ directive tells the compiler under what clock frequency the code is expected to run.

Note: delay code generated by the linker is based on this figure.

Examples:

```
Rem Set 20 MHz clock frequency
#pragma CLOCK_FREQ 20000000
```

#pragma OPTIMIZE

Syntax: *#pragma OPTIMIZE "Flags"*

Elements: **Flags** are the optimization flags also used on the command line.

Purpose: This directive sets new optimization, at function level.
It must be used in the global scope and applies to the function that follows this pragma.

The pragma argument should be enclosed into quotes and is same as argument of the -O compiler command line options.

Empty quotes reset the optimization level previously set by this pragma.

This is the current list of valid optimization flags:

0	no or very minimal optimization
1	regular optimization (recommended)
a	aggressive optimization
p	promotes results of some 16 bit operations to 32 bits

Examples:

```
Rem Use aggressive optimization for subroutine 'foo'
#pragma OPTIMIZE "a"

sub foo()
  ...
end sub
```

BASIC language

Basic is an acronym for Beginners All purpose Symbol Instruction Code. It is true to say that BoostBasic is far more than just for beginners, its a professional programming language that follows the structure of other professional basic languages.

Program Structure

Functions

Every basic program consists of at least one subroutine or function, but normally many.
The difference between functions and subroutines is that a function is able to return some data in the form of a return value to the code that called it.
The return value of a function is the final value assigned to the function name when the function exits.

```
Sub Main()

    Dim res As Integer
    res = Call MyMul

End Sub

Function MyMul() as integer

    MyMul = 10 * 10

End Function
```

Code Documentation

Remarks (comments) can added to the code by prefixing code with Rem statement or ' (apostrophe) .
When Rem is used it must be the only statement on the line, e.g.:

```
Function MyMul() as integer
    Rem This is a comment
    MyMul = 10 * 10 'This code multiplies two numbers - another comment

End Function
```

Multiple Statements Per Line

Multiple statements can appear on one line by separating the statements by a colon (:).
Multiple declarations can appear on the same line by separating them by a comma (,).

Example:

```
Function Foo() as integer

    Dim x As Integer, y As Integer 'Multiple declarations on one line
    x = 10: y = 5 'Multiple statements on one line
    Foo = x * y

End Function
```

Program Entry Point

Once the system initialization is done, the program starts by executing the function name specified in the tool suite setup (main function).
The default entry point is a subroutine called **Main**.

Interrupt Entry Point

Interrupt entry point is a subroutine called **Interrupt**. For PIC18 targets entry point for low priority interrupt is called **Interrupt_low**

Calling and Exiting Functions and Subroutines

One function or subroutine is able to call another. A function call can be coded a number of different ways, they all result in the same code being generated:

Syntax:

[**Call**] *function-name* ([*argument-list*])

The **Call** Statement has the following parts:

Part	Description
Call	Mandatory keyword *(in future releases this keyword will be optional)*.
function-name	The name of the Function or Subroutine (Sub) to be called.
argument-list	Optional argument list that will be passed to the function

When the return value of a function is to be used as part of an expression, the **Call** keywords must be omitted.

Examples:

```
Sub Main()

    Dim val As Integer
    val = Call MyFunc(10) 'call function and use returned value

    Rem call function and do not use returned value - all produce same code

    Call MyFunc(10) 'return value is discarded
    MyFunc (10) 'incorrect, the 'call' keyword was omitted

    Rem call subroutine
    Call MySub(10) 'call subroutine
    MySub (10) 'incorrect, the 'call' keyword was omitted

End Sub

Function MyFunc(x As Integer ) As Integer

    MyFunc = x * 10

End Function

Sub MySub(x As Integer)

    Dim v as integer
    v = x * 10

End Sub
```

A function is exited when the point of execution either reaches the end of a functions body (**End Function** or **End Sub**).
A function can also be exited by using **Exit Sub** or **Exit Function**.

Examples:

```
Sub Main()

    Dim val As Integer
    val = Call MyFunc(1) 'call function and use returned value

    Call MySub(20) 'call subroutine

End Sub

Function MyFunc(x As Integer ) As Integer

    MyFunc = 10
    If x = 1 Then
        Exit Function 'This function can exited at this point
    End If

    MyFunc = 20

End Function

Sub MySub(x As Integer)

    Dim v as integer
    v = x * 10

    If v > 100 Then
        Exit Sub 'This subroutine can be exited at this point
    End If

    v = 0

End Sub
```

Function Arguments

Function arguments can be passed to a function either by value (ByVal)or by reference (ByRef).
When ByVal is used, a copy is taken of the original, so modifying the argument inside the function has no affect on the variable used in the function call.
When ByRef is used, any changes make to the variable will affect the value of the variable used in the function call.
If neither ByRef or ByVal are specified, then the default argument passing mode is ByVal (this is the reverse of some similar style BASICs, but is more efficient on small memory target devices), except for arrays which are passed ByRef.

Example 1:

```
Rem Example of Function arguments past by reference and by value.

Sub main()

    Dim v As Integer

    v = 10
    Call Inc1( v )
    Rem v will now have a value of 11


    v = 10
    Call Inc2( v )
    Rem v will be unchanged, i.e. has a value of 10

    Rem loop forever
    do while( 1 )
    loop

End Sub

Sub Inc1( ByRef x As Integer )

     x = x + 1

End Sub

Sub Inc2( ByVal x As Integer )

     x = x + 1

End Sub
```

Example 2:

```
Rem Example of an array being passed as a function argument

Sub main()

    Dim v( 10 ) As Integer

    v( 0 ) = 10
    v( 1 ) = 20

    Call Inc1( v, 0 ) ' v( 0 ) will have value of 11
    Call Inc1( v, 1 ) ' v( 1 ) will have value of 21

    Rem loop forever
    do while( 1 )
    loop

End Sub

Sub Inc1( x() As Integer, y As Integer )

     x( y ) = x( y ) + 1

End Sub
```

Variable Data Types

Variables can have the following data type

Bit	1 bit	0 or 1
Boolean	1 bytes	**True** or **False**
Byte	1 byte	0 to 255
Char	1 byte	-128 to 127
Word	2 bytes	0 to 65535
Integer	2 bytes	-32,768 to 32,767
Dword (double word)	4 bytes	0 to 4,294,967,296
Long (long integer)	4 bytes	-2,147,483,648 to 2,147,483,647
String (variable-length)	x bytes + string length	Dependant on string length
User-defined (using **Type**)	Number required by elements	The range of each element is the same as the range of its data type.

Variable Declarations

Syntax:
Dim *varname* [@ fixed-*address]* ***As*** [*volatile*] **Byte**

Variable names must begin with an alphabetic character, must be unique within the same scope, can't be longer than 255 characters, and can't contain an embedded period or type-declaration character.

Variable can be declared inside functions or outside functions. Variables declared outside a function are global variables, this means any function can access them. Variables declare inside a function are local variables, the can only be accessed by code inside the function. Local variables are also declared on the stack, this means that their value is lost between one call of a function and the next

When a variable is declared it is give a default value. Numerical variables are give a value of 0, strings are empty and boolean's are set to false.

Variables are declared (Dimensioned) in the following way:

```
Dim x As Byte ' Declares a single byte variable called x
Dim y(10) As Byte ' Declares an array of bytes called y
```

When a variable is declared as static, then its data is preserved between one call of a function and the next. Every time the example function MyFunc is called it returns a value that is increase by 1:

```
Function MyFunc()

     Static x As Integer
     x = x + 1
     MyFunc = x

End Sub
```

Register mapped variables

It can be desirable to give variables fixed addresses. This allows variables to be associated with the hardware registers of peripheral devices. The address value can be specified in hexadecimal or decimal

Example:

```
Dim mytimer @ 0x01 As Volatile Byte ' map variable to PIC16 tmr0
Dim myport @ 0x06 As Volatile Byte ' map variable to PIC16 portb
Dim myvar @ 32 As Byte ' map variable to PIC16 fixed RAM location
```

Many such declarations already exist in header files associating hardware registers with variables. These declarations are automatically included in a project when SourceBoost IDE is used as the development environment.

Volatile type specifier

The **volatile** type specifier should be used with variables that:

a) Can be changed outside the normal program flow, and

b) Should **not** receive intermediate values within expressions.

For example, if a bit variable is mapped to a port pin, it is a good programming practice to declare such variable as *volatile*.

Code generated for expressions with *volatile* variables is a little longer when compared to 'regular' code:

String Declarations

Fixed length strings are declared using the following syntax:

Dim *mystring* **As String (***constant-string-length*)

The string declarations has the following parts:

Part	Description
Dim	keyword.
mystring	The name of used to refer to this string object.
As String	The string data type.
constant-string-length	An argument that sets the fixed length of this string.

Example:

```
Sub Main()

     Dim msg1 As String (20) 'declare string of length 20 characters
     Dim msg2 As String (20) 'declare string of length 20 characters
     Dim msg3 As String (20) 'declare string of length 20 characters

     msg1 = "hello" ' assign characters to string
     msg2 = "there" ' assign characters to string

     msg3 = msg1 + " " + msg2 ' concatenate strings, msg will contain
"hello there"

End Sub
```

Note: When declaring **Strings**, only make them as long as required. Making them longer will use more memory that then cannot be for other data items in your program.

If...Then...Else Statement

Conditionally executes a group of statements, depending on the value of an expression.

Syntax

If *condition* **Then** [*statements*] [**Else** *elsestatements*]

Or, you can use the block form syntax:

If *condition* **Then**
[*statements*]

[**ElseIf** *condition-n* **Then**
[*elseifstatements*] ...

[**Else**
[*elsestatements*]]

End If

The **If...Then...Else** statement syntax has these parts:

Part	***Description***
condition	Required. One or more of the following two types of expressions:
	A numeric expression or string expression that evaluates to **True** or **False**. If *condition* is Null, *condition* is treated as **False**.
statements	Optional in block form; required in single-line form that has no **Else** clause. One or more statements separated by colons; executed if *condition* is **True**.
condition-n	Optional. Same as *condition*.
elseifstatements	Optional. One or more statements executed if associated *condition-n* is **True**.
elsestatements	Optional. One or more statements executed if no previous *condition* or *condition-n* expression is **True**.

If..then..else example code:

```
Sub Main()

     Dim x As Integer
     Dim y As Integer
     Dim z As Integer

     x = 0
     y = 2
     z = 3

     ' simplest conditional statement
     If x <> y Then
          z = 4
     End If

     ' simple with else
     If x = 1 And y = 3 Then
          z = 1
     Else
          z = 2
     End If

     ' use of elseif
     If x = 10 Then
          y = 1
     ElseIf x = 5 Then
          y = 2
     ElseIf x = 7 Then
          y = 3
     Else
          y = 4
     End If

End Sub
```

For Next

Repeats a group of statements a specified number of times.

Syntax

For *counter* **=** *start* **To** *end* [**Step** *step*]
[*statements*]
[**Exit For**]
[*statements*]

Next [*counter*]

The **For...Next** statement syntax has these parts:

Part	*Description*
counter	Required. Numeric variable used as a loop counter. The variable can't be a Boolean or an array element.
start	Required. Initial value of *counter*.
end	Required. Final value of *counter*.
step	Optional. Amount *counter* is changed each time through the loop. If not specified, *step* defaults to one.
statements	Optional. One or more statements between **For** and **Next** that are executed the specified number of times.

Remarks

The *step* argument can be either positive or negative.
The loop terminates when the loop *counter* exceeds the *end* value (if *step* is positive this is when *counter* > *end*, if *step* is negative this is when *counter* is < *end*).

Example:

```
Function ReadChkSums() As Byte

    Dim sum As Integer
    Dim i As Integer

    ' Read ten values from Addresses 50, 45, 40, ... 5, 0
    For i = 50 To 0 Step -5
        sum = sum + Call ReadData(i)
    Next

    ReadChkSums = sum

End Function
```

Do While Loop

Repeats a block of statements **while** a condition is **True**.

Syntax

Do While *condition*
[*statements*]
[**Exit Do**]
[*statements*]

Loop

Or, you can use this syntax:

Do
[*statements*]
[**Exit Do**]
[*statements*]

Loop While *condition*

The **Do Loop** statement syntax has these parts:

Part	***Description***
condition	Optional. Numeric expression or string expression that is **True** or **False**. If *condition* is Null, *condition* is treated as **False**.
statements	One or more statements that are repeated while, or until, *condition* is **True**.

Exit Do allows the loop to be exited without the **While** *condition* becoming true. When **Exit Do** is executed the loop is immediately executed, no more statements within the loop get executed.

While may be substituted with **Until,** the loop then continues until the condition is **True**. This means that **While** *condition* has the same meaning as **Until Not** *condition*.

Examples of **Do While Loop** usage:

```
Function CalcFactorial(factorial As Integer) As Integer

    Rem Calculate the factorial of a number
    Rem e.g. 4 factorial = 1 * 2 * 3 * 4 = 24

    Dim cnt As Integer
    Dim val As Integer

    cnt = 2
    val = 1
    Do While cnt <= factorial

        val = val * cnt
        cnt = cnt + 1

    Loop
    CalcFactorial = val

End Function

Function WaitForKey(timeOutCnt As Integer) As Byte

    Dim k As Byte
    Dim cnt As Integer

    Do
        ' Get the current key pressed
        k = Call GetKey()

        ' Terminate after some many checks, a key may never be pressed!
        If cnt > timeOutCnt Then
            Exit Do
        End If
        cnt = cnt + 1

    Loop Until k

    WaitForKey = k

End Function
```

Select Case Statement

Executes one of several groups of statements, depending on the value of an expression.

Syntax

Select Case *testexpression*
[**Case** *constant-n*
[*statements-n*]] ...
[**Case Else**
[*elsestatements*]]

End Select

The **Select Case** statement syntax has these parts:

Part	*Description*
testexpression	Required. Any numeric expression.
constant-n	Required if a **Case** appears. Delimited list of one or more of the following forms: *constant-n*, *constant-n1* **To** *constant-n2*.
statements-n	Optional. One or more statements executed if *testexpression* matches any part of *constant-n.*
elsestatements	Optional. One or more statements executed if *testexpression* doesn't match any of the **Case** clause.

Example:

```
Function GetLedBarPattern(patternNumb As Integer) As Byte

    ' converts a number into an patter we can use to drive an LED bar graph
    Select Case patternNumb
        Case 0
            GetLedBarPattern = 0
        Case 1
            GetLedBarPattern = 1
        Case 2
            GetLedBarPattern = 3
        Case 3
            GetLedBarPattern = 7
        Case 4
            GetLedBarPattern = 15
        Case 5
            GetLedBarPattern = 31
        Case 6
            GetLedBarPattern = 63
        Case 7
            GetLedBarPattern = 127
        Case 8
            GetLedBarPattern = 255
        Case 9, 10
            GetLedBarPattern = 240
        Case 11 To 50
            GetLedBarPattern = &H5A ' show special overload pattern
        Case Else
            GetLedBarPattern = &HA5 ' show special overload pattern
   End Select

End Function
```

Enum Statement

Declares a type for an enumeration.

Syntax

Enum *name*

membername [= *constantexpression*]

membername [= *constantexpression*]

. . .

End Enum

The **Enum** statement has these parts:

Part	*Description*
name	Required. The name of the **Enum** type. The *name* must be a valid identifier and is specified as the type when declaring variables or parameters of the **Enum** type.
membername	Required. A valid identifier specifying the name by which a constituent element of the **Enum** type will be known.
constantexpression	Optional. Value of the element (evaluates to a **Long**). If no *constantexpression* is specified, the value assigned is either zero (if it is the first *membername*), or 1 greater than the value of the immediately preceding *membername*.

Example:

```
Enum Colors
    Red = 1
    Blue = 2
    Green = 3
End Enum

Function IsBlue(colorVal As Integer) As Boolean

    If colorVal = Blue Then
        IsBlue = True
        Exit Function
    End If

    IsBlue = False

End Function
```

Note: An enum cannot be declared inside a function.

Type Statement

The Type statement is used to generate user defined data types.

Syntax

Type *varname*
elementname [**(**[*arraySize*]**)**] **As** *type*
[*elementname* [**(**[*arraySize*]**)**] **As** *type*]
. . .
End Type

Example:

```
Type MyData
    x As Integer
    y As Byte
    results(10) As Integer
End Type

Sub Main()

    Dim z As MyData
    z.x = 1000 'Assigning member x of user defined data structure
    z.y = 200 'Assigning member x of user defined data structure
    z.results(9) = 53 'Assigning member results, element 9
                      '(the last element)

End Sub
```

Comparison Operators

Used to compare expressions.

Syntax

result **=** *expression1 comparisonoperator expression2*

Comparison operators have these parts:

Part	***Description***
result	Required; any numeric variable.
expression	Required; any expression.
comparisonoperator	Required; any comparison operator.
object	Required; any object name.
string	Required; any string expression.
pattern	Required; any string expression or range of characters.

The following table contains a list of the comparison operators and the conditions that determine whether *result* is **True**, **False**:

Operator	***True if***	***False if***
< (Less than)	*expression1 < expression2*	*expression1 >= expression2*
<= (Less than or equal to)	*expression1 <= expression2*	*expression1 > expression2*
> (Greater than)	*expression1 > expression2*	*expression1 <= expression2*
>= (Greater than or equal to)	*expression1 >= expression2*	*expression1 < expression2*
= (Equal to)	*expression1 = expression2*	*expression1 <> expression2*
<> (Not equal to)	*expression1 <> expression2*	*expression1 = expression2*

Note: Only variables of the same data type can be compared.

Caparison Example:

```
Function TextToNumb( msg As String ) As Integer

    Rem Function to turn numbers in the form of words to numerical values
    Rem Zero is returned if the number is not recognized

    Dim isEmpty As Boolean

    isEmpty = (msg = "")
    TextToNumb = 0 ' Set default return value

    If isEmpty Then
        TextToNumb = -1
    End if

    If msg = "One" Then
        TextToNumb = 1
    End If

    If msg = "Two" Then
        TextToNumb = 1
    End If

    If msg = "Three" Then
        TextToNumb = 1
    End If

End Sub

Function GetMax( v1 As Integer, v2 As Integer ) As Integer

    Rem Function that returns the biggest of two numbers

    If v1 >= v2 Then
        GetMax = v1
        Exit Function
    End if

    GetMax = v2

End Sub
```

Operators

Operator	Operation description	Example

^	Bitwise Exclusive Or	`Dim a As Byte, b As Byte`
		`a = 255`
		`b = a ^ 7 ' returns 248`

|	Bitwise Or	`Dim a As Byte, b As Byte`
		`a = 255`
		`b = a | 7 ' returns 255`

&	Bitwise And	`Dim a As Byte, b As Byte`
		`a = 255`
		`b = a | 7 ' returns 7`

*	Multiply. The result is the same as the most precise number promoted, e.g. byte is promoted is integer and integer is promoted to long. Long and Single do not get promoted.	`Dim x As Integer` `x = 10 * 3 ' returns 30`
\	Integer divide.	`Dim x As Single` `x = 10 \ 3 ' returns 3`

Mod	Divide one number by another and return the remainder. If either of the numbers is a floating point they are rounded to an integer before the divide.	`Dim x As Integer`
		`x = 12 Mod 4.3 ' returns 0`
		`x = 10 Mod 3 ' returns 1 .`

+	Sum two numbers, or concatenate two strings.	`Dim msg As String`
		`msg = "hel" + "lo" ' creates string "hello"`
		`Dim z As Integer`
		`z = 10 + 20 ' returns 30`

-	Subtract two numbers.	`Dim z As Integer` `z = 20 - 15 ' returns 15`

Or	Logical Or	`Dim x As Byte, y As Byte, z As Byte`
		`x = 15`
		`y = 3`
		`z = x Or y ' returns 1`

And	Logical And	`Dim x As Byte, y As Byte, z As Byte`
		`x = 15`
		`y = 3`

		z = x And y ' returns 1 x = 15 y = 0 z = x And y ' returns 0
<<	Left shift	Dim x As Byte, y As Byte x = 10 y = x << 2 ' returns 40
>>	Right shift	Dim x As Byte, y As Byte x = 255 y = x >> 2 ' returns 63
!	Logical Not	Dim x As Char, y As Char, z As Char x = 10 : y = 0 z = !x ' returns 0 z = !y ' returns 1
~	Bitwise Not	Dim x As Byte, y As Byte x = 10 y = ~x ' returns 245
- (negate)	Negate	Dim x As Char, y As Char x = 10 y = -x ' returns -10

Concatenation Operators

Concatenation of character strings can be performed using either of the following operator: & or +.

Syntax

string1 **+** *string2* [**+** *stringn*]

or

string1 **&** *string2* [**&** *stringn*]

Example:

```
Function CombineStrings( s1 As String, s2 As String, s3 As String ) As String

    CombineStrings = s1 & s2 & s3
    CombineStrings = s1 + s2 + s3 ' does the same as the line above!!!

Sub Function
```

Bit Access

Separate bits of a variable can be accessed using the dot '.' operand

Example:

```
Dim a As Integer
a.2 = 1 ' set bit 2 of the variable a
```

Inline assembly

Use the asm or _asm operators to embed assembly into C code.

Bank switching and code page switching code should NOT be added to inline assembly code. The linker will add the appropriate Bank switching and code page switching code.

asm

Code will be affected as follows:

- Bank switching added automatically.
- Code page switching added automatically.

_asm

Code will be affected as follows:

- Bank switching added automatically.
- Code page switching added automatically.
- Other optimizations applied (including dead code removal).

Assembly operators can be used in single line or multi line modes. Single line mode operates on one assembly instruction that follows the asm or _asm operators on the same line:

```
asm nop
```

In multi line mode assembly instructions are enclosed into curly braces that should be placed on different than assembly code lines:

```
asm
{
      nop
      nop
      nop
}
```

Variable Referencing in asm

To refer to a C variable from inline assembly, simply prefix its name with an underscore '_'. If a C variable name is used with the 'movlw' instruction, the address of this variable is copied into W.
Labels are identified with a trailing semicolon ':' after the label name.

Constants in asm

Inline assembly supports decimal and hexadecimal constants only:

XXXX	decimal number, where X is a number between 0 and 9
0xXXXX	hexadecimal number, where X is a number between 0 and 9 or A and F
0bXXXX	binary number, where X is a number between 0 and 1

Inline assembly example 1

```
' Example showing use of bit tests and labels in inline assembly

Sub foo()

 Dim i As Char
 Dim b As Char
 i = 0
 b = 12

 asm
 {
  start:
  btfsc _i, 4
  goto  end
  btfss _b, 0
  goto  iter

  iter:
  movlw 0
  movwf _b
  end:
 }

End sub
```

Inline assembly example 2

```
' Example for PIC18F8720 target showing how to access bytes of
' integer arguments

Function GetTmr1Val() As Integer

  Dim x As Integer
  asm
  {
    movf _tmr1h, W
    movwf _x+1  ; write to high byte of variable x
    movf _tmr1l, W
    movwf _x    ; write to low byte of variable x
  }

  GetTmr1Val = x

End Function
```

Inline assembly example 3

```
' Note: This code may not work as expected if the data structure
' is modified causing member count2 to have a different offset.

Type Stats
     count0 As Integer ' stored in bytes 0 & 1
     count1 As Byte  ' stored in byte 2
     count2 As Integer     ' stored in bytes 3 & 4
End Type

Dim myStats As Stats

Sub AddCount2()

  Dim x As Integer
  asm
  {
    movf _myStats+3, W
    addlw 0x01
    movwf _myStats+3
    btfsc _status, C
    incf _myStats+4,F
  }
End Sub
```

Setting Device Configuration Options

In order for a program to be able to run on a target device the device configuration options need to be correctly set. For example having the wrong oscillator configuration setting may mean that the device has no clock, making it impossible for any code to be executed. Configuration data is set using the pragma directive: *#pragma DATA.*

Configuration options typically control:

- Oscillator configuration
- Brown out reset
- Power up reset timer
- Watchdog configuration
- Peripheral configurations
- Pin configurations
- Low voltage programming
- Memory protection
- Table read protection
- Stack overflow handling

The exact configuration options available depend on exactly which device is being used. The PIC18 devices have many more configuration options that the PIC16/PIC12 devices.

Configuration Example 1:

```
Rem Configuration for PIC16F874A
#pragma DATA _CONFIG, _CP_OFF & _PWRTE_OFF & _WDT_OFF & _HS_OSC & _LVP_OFF
```

Configuration Example 2:

```
Rem Configuration for PIC18F452
#pragma DATA    _CONFIG1H, _OSCS_OFF_1H & _HS_OSC_1H
#pragma DATA    _CONFIG2L, _BOR_ON_2L & _BORV_20_2L & _PWRT_OFF_2L
#pragma DATA    _CONFIG2H, _WDT_OFF_2H & _WDTPS_128_2H
#pragma DATA    _CONFIG3H, _CCP2MX_ON_3H
#pragma DATA    _CONFIG4L, _STVR_ON_4L & _LVP_OFF_4L & _DEBUG_OFF_4L
#pragma DATA    _CONFIG5L, _CP0_OFF_5L & _CP1_OFF_5L & _CP2_OFF_5L & _CP3_OFF_5L
#pragma DATA    _CONFIG5H, _CPB_OFF_5H & _CPD_OFF_5H
#pragma DATA    _CONFIG6L, _WRT0_OFF_6L & _WRT1_OFF_6L & _WRT2_OFF_6L & _WRT3_OFF_6L
#pragma DATA    _CONFIG6H, _WRTC_OFF_6H & _WRTB_OFF_6H & _WRTD_OFF_6H
#pragma DATA    _CONFIG7L, _EBTR0_OFF_7L & _EBTR1_OFF_7L & _EBTR2_OFF_7L & _EBTR3_OFF_7L
#pragma DATA    _CONFIG7H, _EBTRB_OFF_7H
```

You can find the defined configuration options for a given device by looking in the target devices BoostBasic header file (PIC18XXXX.BAS and PIC16XXXX.BAS) which can be found in the installation directory (typically "C:\Program File\SourceBoost\include\basic"). Its also worth looking at relevant Microchip data sheet to find the exact function of each option.

Initialization of EEPROM Data

It is often desirable to program the PIC on board EEPROM with initial data as part of the programming process. This initial data can be included in the source code. EEPROM initialization data is set using the pragma directive: *#pragma DATA.*

Example:

```
Rem Initializes EEPROM with data: 0C 22 38 48 45 4C 4C 4F 00 FE
#pragma DATA _EEPROM, 12, 34, 56, "HELLO", 0xFE
```

Using BoostC libraries in BoostBasic

It is possible to use BoostC libraries in BoostBasic. There are two things that need to be done in order to do this:

1. Add a function External declaration.
2. Add the appropriate library to the project.

In order to generate the BoostBasic External declaration the BoostC equivalent types need to be know:

Equivalent data types Table	
BoostC Type	**BoostBasic Type**
bit	Bit
bool	Boolean
char	Byte
signed char	Char
unsigned short	Word
signed short	Integer
unsigned int	Word
signed int	Integer
unsigned long	Dword
signed long	Long
char *	Byte array

Function External Declaration

This declaration informs the compiler that such a function exists.

Syntax:

Extern *function-name*([*argument-list*])

Example:

BoostC function prototype:
char sprintf(char *, char*, unsigned int);

BoostBasic External function declaration:
Extern Function sprintf(buffer() as byte, format() as byte, val as word) as Byte

```
' Calling a BoostC function from within BoostBasic
'
' Declare the external function
Extern Function sprintf( buffer() As Byte, format() As Byte, val As Word ) As Byte

Sub main()

      Dim msg As String( 21 )
      Dim v As Integer

      v = 1234
      Call sprintf( msg, "my value is %d", v )

      ' loop forever
      Do While( 1 )
      Loop

End Sub
```

PC System Requirements

In order to install and run the Compiler/SourceBoost Integrated Development Environment, a PC with the following specification is required:

Minimum System Specification
Microsoft Windows 98/ME/NT/2000/XP,
Adobe Reader and a web browser (to allow access to help files and manuals).
Pentium Processor or equivalent,
128MB of RAM,
CD ROM Drive,
80MB of disk space,
16Bit Color display Adapter at 800x600 Resolution.

Recommended System Specification
As the Minimum System Specification, plus:
2.0GHz (or faster) Processor,
512MByte (or more) RAM,
16Bit Color display Adapter at 1024x768 Resolution (or higher).

Technical support

For example projects and updates please refer to our website: http://www.sourceboost.com

We operate a forum where technical and license issue problems can be posted. This should be the first place to visit:
http://forum.sourceboost.com

Licensing Issues

If you have licensing issues, then please send a mail to:
support@sourceboost.com

General Support

For general support issues, please use our support forum:
http://forum.sourceboost.com

We are always pleased to hear your comments, this helps us to satisfy your needs. Post your comments on the SourceBoost Forum or send an email to:
support@sourceboost.com

Legal Information