BoostC C Compiler

for PICmicro

Reference Manual

V1.18

Index

This page is intentionally left blank.

BoostC compiler

Introduction

Thank you for choosing BoostC. **BoostC** is our next generation C compiler that works with PIC16, PIC18 and some PIC12 processors.

This ANSI C compatible compiler supports features like source level symbolic debugging, signed data types, structures/unions and pointers.

The **BoostC** compiler can be used within our **SourceBoost IDE** (Integrated Development Environment), or it can be integrated into Microchip MPLAB.

BoostC Compiler specification

Base Data types

Size	Type name	Specification
1 bit	*bit, bool*	boolean
8 bit	*char*	signed, unsigned
16 bit	*short, int*	signed, unsigned
32 bit	*long*	signed, unsigned

Special Data types

- **Single bit** - single bit data type for efficient flag storage.
- **Fixed address** - fixed address data types allow easy access to target device registers.
- **Read only** - code memory based constants.

Special Language Features

- References as function arguments.
- Function overloading.
- Function templates.

Code Production and Optimization Features

- **ANSI 'C' compatible** - Makes code highly portable.
- Produces optimized code for both PIC16 (14bit core) and PIC18 (16bit core) targets.
- **Support for Data Structures and Unions** - Data structures and arrays can be comprised of base data types or other data structures. Arrays of base data types or data structures can be created.
- **Support for pointers** - pointers can be used in "all the usual ways".
- **Inline Assembly** - Inline assembly allows hand crafted assembly code to be used when necessary.

- **Inline Functions** - Inline functions allows a section of code to be written as a function, but when a reference is made to it the inline function code is inserted instead of a function call. This speeds up code execution.
- **Eliminates unreachable (or dead) code** - reduces code memory usage.
- **Removal of Orphan (uncalled) functions** - reduces code memory usage.
- **Minimal Code Page switching** - code where necessary for targets with multiple code pages.
- **Automatic Banks Switching for Variables** - allows carefree use of variables.
- **Efficient RAM usage** - local variables in different code sections can share memory. The linker analyzes the program to prevent any clashes.
- **Dynamic memory management**.

Debugging features

- **Source Level and Instruction Level Debugger** - linker Generates COF file output for source level debugging under SourceBoost D[ebugger.]
- **Step into, Step over, Step out and Step Back** – these functions operate both at source level and instruction level.
- **Multiple Execution Views** - see where the execution of the code is at source level and assembly level at the same time.
- **Monitoring variables** - variables can be added to the watch windows to allow there values to be examined and modified. There is no need to know where a variable is stored.

Full MPLAB integration

- 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 environment.
- Source level debugging and variable monitoring using:
    - **MPLAB simulator;**
    - **MPLAB ICD2;**
    - **MPLAB ICE2000.**

Librarian

- Allows generation of **library files** - this simplifies management and control of regularly used, shared code.
- Reduce compilation time - using library files reduces compilation time.

Code Analysis

- **Call tree view** - **SourceBoost IDE** can display the function call tree.
- **Target Code Usage** - From the complete program, down to Function level the code space usage can be viewed in SourceBoost IDE.
- **Target RAM Usage** - From the complete program, down to Function level the RAM usage can be examined and reviewed in SourceBoost IDE.

Installation

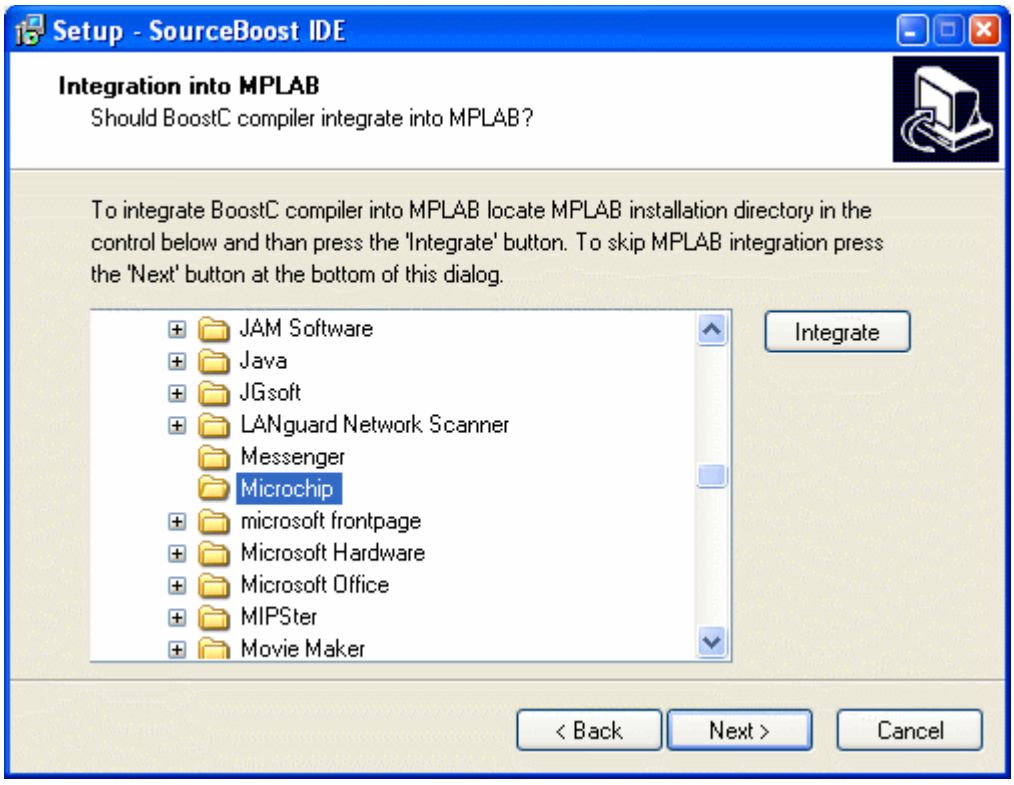
The BoostC compiler cannot be downloaded or installed on its own. BoostC is part of the SourceBoost software package that includes the SourceBoost IDE and other Language Suites. It is available for download from our site http://www.sourceboost.com

When you buy a license, you will activation code(s) and detailed instructions on how to activate the compiler and other tools you have licensed.

To install SourceBoost IDE and BoostC on your system, please follow these simple steps:

- Execute the installer sourceboost.exe and follow on-screen directions.

- Please pay attention to the integration dialog:

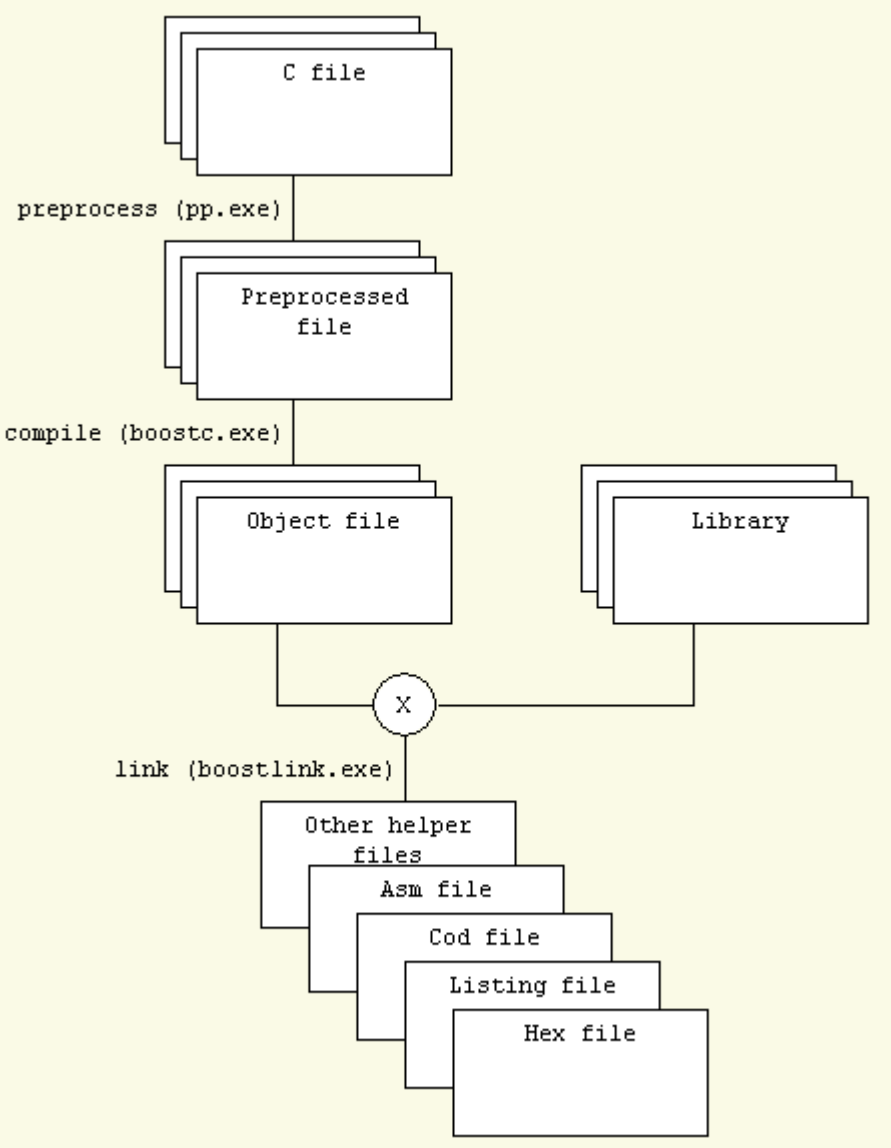


  To integrate BoostC with MPLAB, choose the correct Microchip installation directory, then click on "Integrate" before stepping to the next installation wizard dialog.

- The rest of the installation process is straightforward. At the end, SourceBoost IDE is ready to be used on your system. Should any difficulty arise, please double check your system configuration and mail all details to support@sourceboost.com

- **Please note:** if the installation step "MPLAB Integration" is skipped, the necessary MPLAB integration files will be installed to the \mplab subdirectory of the chosen SourceBoost installation directory. These files can always be manually copied to the correct location – please see the "MPLAB Integration" section later in this manual.

Compilation model and toolchain



Preprocessor

The preprocessor **pp.exe** is automatically invoked by the compiler.

Compiler

There are actually two separate compilers: one for pic16 and one for pic18 targets.

When you work under SourceBoost IDE, there is no need to specify which one to use: the IDE picks the correct compiler based on the selected target.

The output of the compiler is one or more **.obj** files, that are further processed by **librarian** or **linker**, in order to get a **.lib** or **.hex** file.

Linker

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

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 changes project target to library instead of hex file.

To create a target independent library, include *boostc.h* instead of *system.h* into the library sources. This way no target specific information (like target dependent constants or variables mapped to target specific registers) is included into the library. Note that this is the only case in which *system.h* does not be included into the code.

Differences with C2C compilation model

The main difference between **BoostC** and our previous generation **C2C** compiler is that the latter had a built-in linker and created an **.asm** file needing to be assembled using an external assembler (like MPASM), while the **BoostC** toolsuite doesn't need any external tools and directly generates the target **.hex** file.

Another difference is in how compilers handle read-only variables located in code memory. **BoostC** uses the special data type specifier '*rom*', while **C2C** placed any variable defined as '*const*' into code memory.

MPLAB integration

BoostC C 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: in case the installation step "MPLAB Integration" was skipped, the files in the <SourceBoost>\mplab directory can be manually copied into

<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 **BoostC** 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

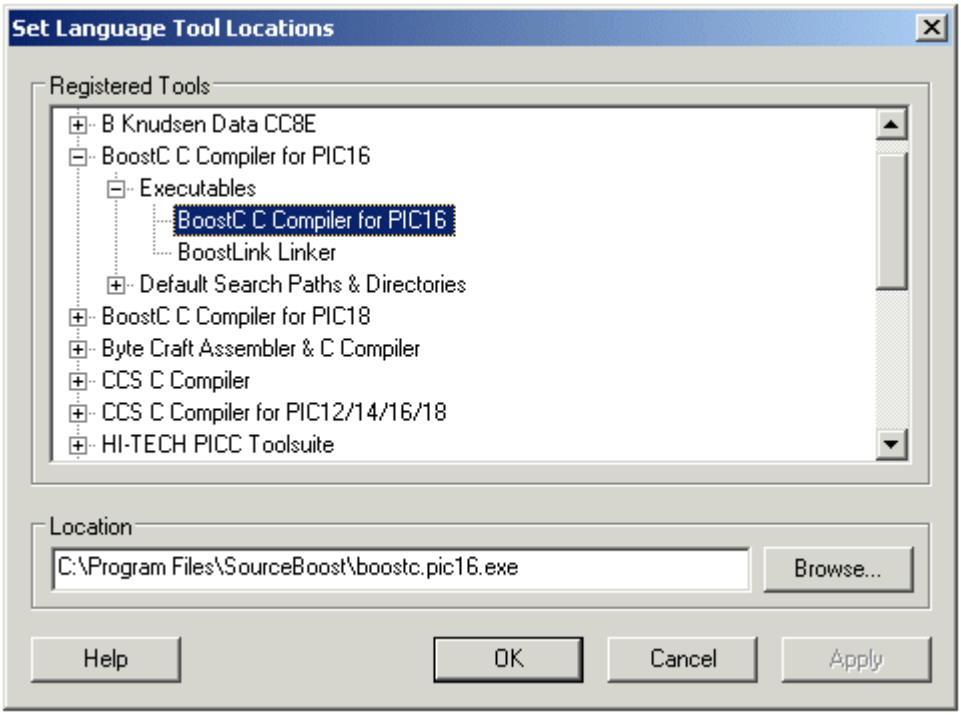
Note: this process only needs to be performed once.

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

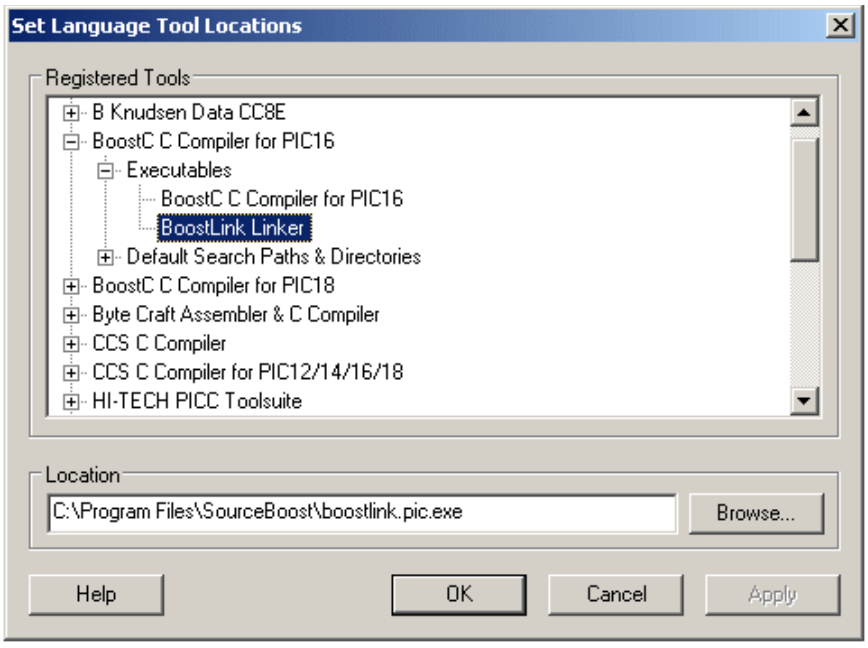
1. Start MPLAB IDE.

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

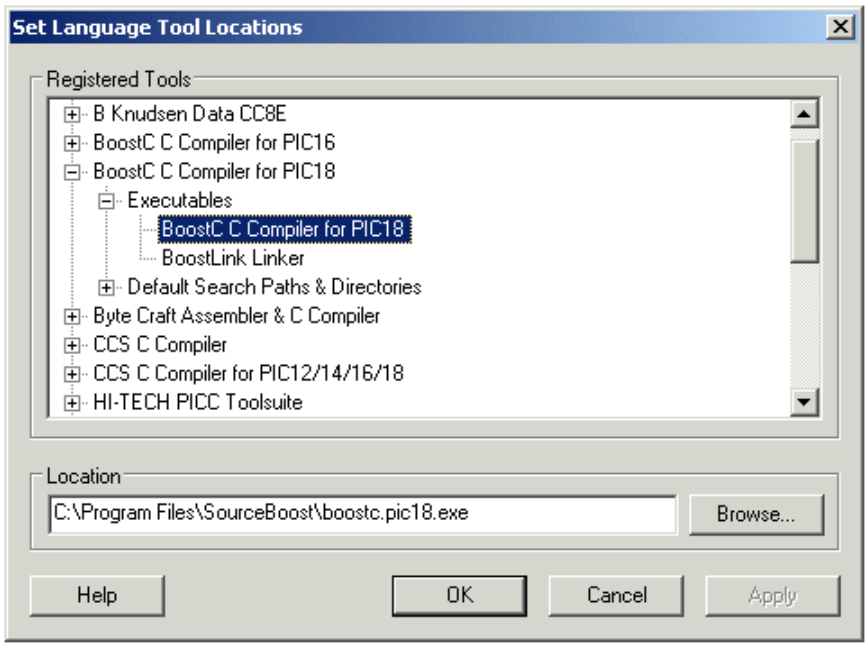
3. Set **BoostC C** compiler for PIC16 location:



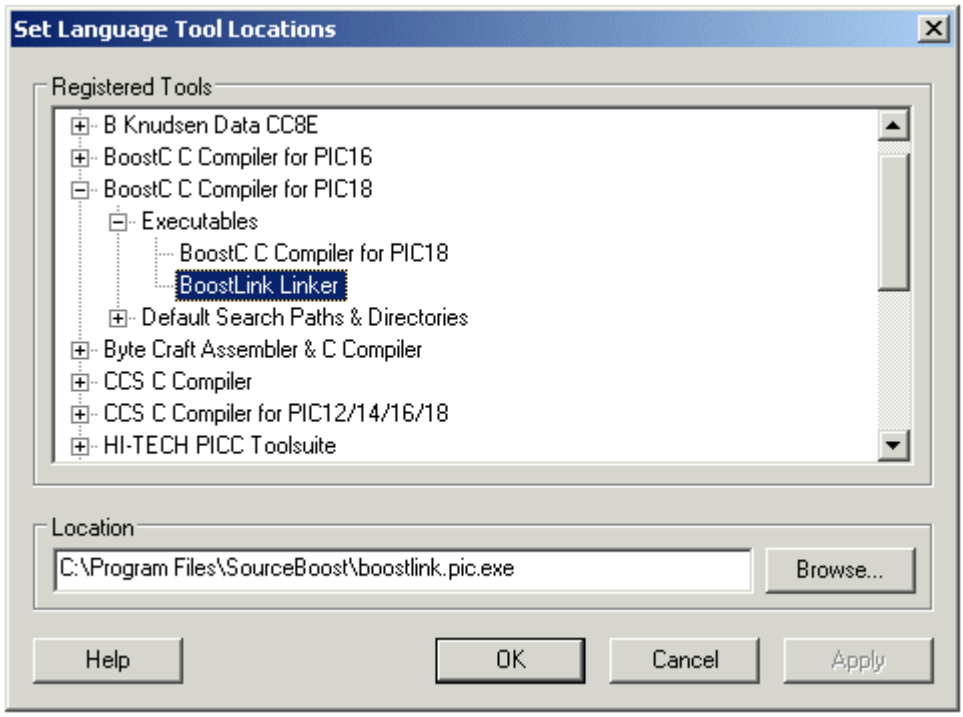
4. Now set **BoostLink** Linker location:



5. Set **BoostC C** compiler for PIC18 location:



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

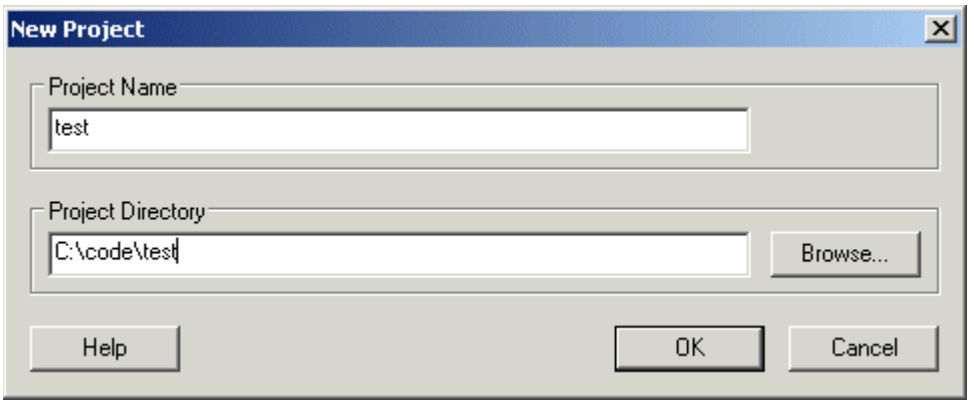


Creating a project under MPLAB IDE

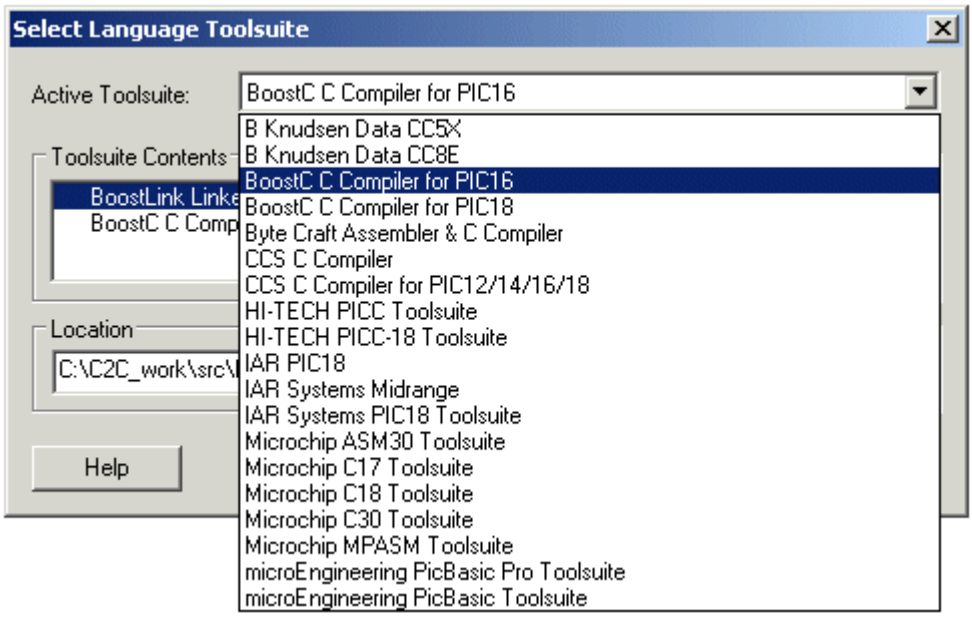
Before attempting to do this, please ensure that the "**Setting the MPLAB language tool locations"** process illustrated in the above section has been successfully performed.

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

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

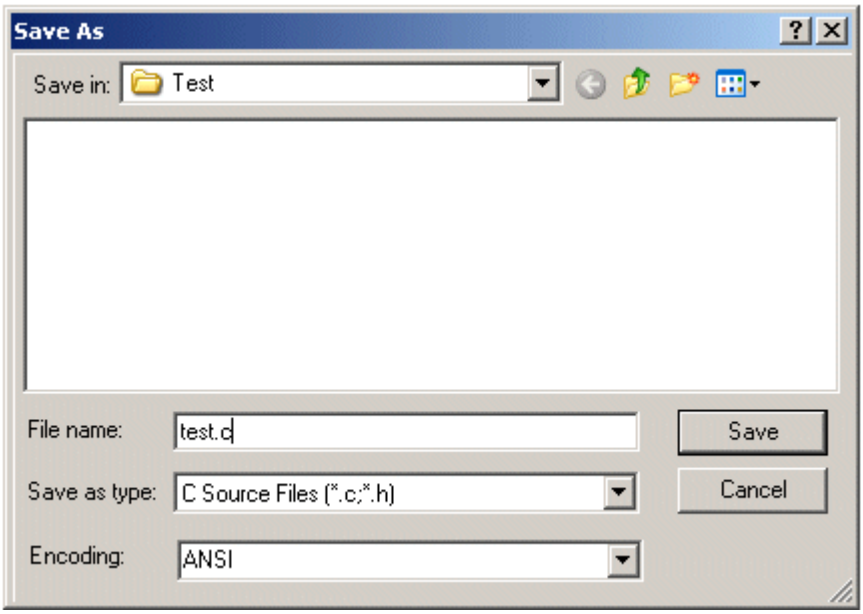


2. Menu **Project ➡ Select Language Toolsuite**. Select the **BoostC C** Compiler for PIC16.

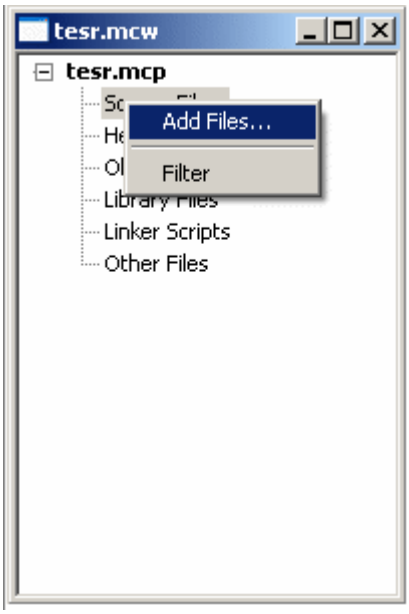


3. Menu **File ➡ New**. Type code into the Untitled window.
   Note: If you already have Source Files, steps 2 and 3 can be skipped.

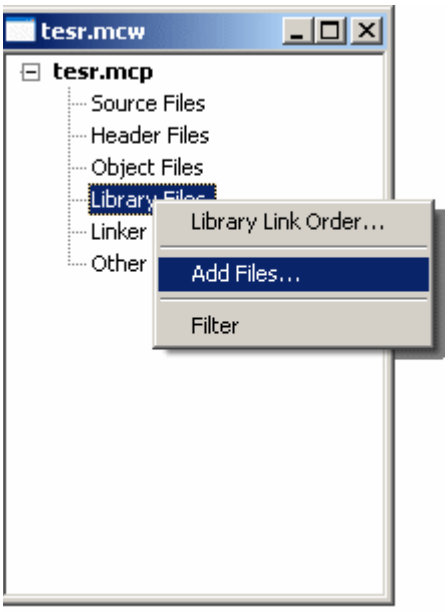
4. Menu **File ➡ Save As**. Locate the project folder using the Save As dialog box.



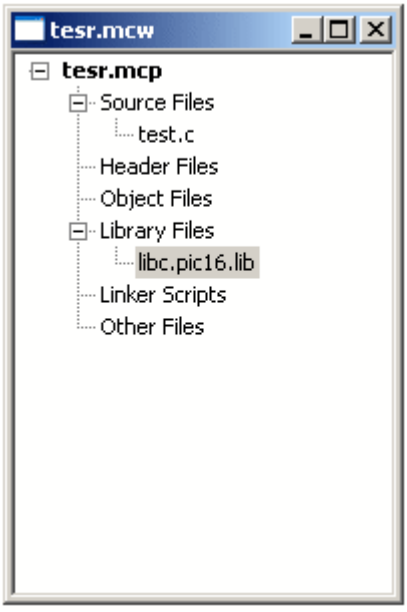
5. Add the test.c source file to the project by right clicking on Source Files in the project tree – as shown below.



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.

For ease of project browsing, you can also add the project header files to the project tree in the same way as the source files where added.

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.h* header file must be included in the source code, eg:

```
#include <system.h>
#include <icd2.h>          // allocates RAM used by ICD2

void main()
{
    while(1);
}
```

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.

Command line options

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

BoostC command line

```
BoostC Optimizing C Compiler Version 6.xx
http://www.sourceboost.com
Copyright(C) 2004-2006 Pavel Baranov
Copyright(C) 2004-2006 David Hobday

Licensed to <license info>

Usage: boostc.pic16.exe [options] files

Options:
  -t name    target processor (default name=PIC16F648A)
  -On        optimization level (default n=1)
             n=0 - optimization turned off
             n=1 - 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
```

Optimization

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

Optimize flags:

-O0 no or very minimal optimization

-O1 regular 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 6.xx
http://www.sourceboost.com
Copyright(C) 2004-2006 Pavel Baranov
Copyright(C) 2004-2006 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)

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

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. This option must be used when using **Novo RTOS**. Where possible the hardware stack is used for efficiency. By specifying the amount of hardware stack to use for main and tasks (Novo tasks), ISR (interrupt service routine) and low priority ISR (PIC18 only), allows control over when the software call stack is use 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. If no hardware stack depths are specified, then the software stack is only used in functions that contain or call functions that contain a **Novo RTOS** Sys_Yield() function.

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.

-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
}
```

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

SourceBoost IDE

The **SourceBoost IDE** is thoroughly covered in a separate manual.

Preprocessor

The **pp.exe** preprocessor is automatically invoked by the compiler. It executes a series of parametrized text substitutions and replacement (macro processing), besides evaluating special directives.

All preprocessor directives start with a '#'. Non standard directives are always contained in statements with a leading ANSI keyword *#pragma*, so to avoid potential conflicts when porting code to other compilers and/or with advanced source analysis tools (lint, static checkers, code formatters, flow analyzers and so on).

Directives

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

#include

#define

#undef

#if

#else

#endif

#ifdef

#ifndef

#error

#warning

These directives are individually explained in the following pages.

#include

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

Elements: **filename** is any valid PC filename. It may include standard drive and path information.
In the event no path is given, the following applies:

a) If filename appears between "", the directory of the projects is searched first.

b) If the delimiters <> are used, only the IDE *include path list* is searched for filename.

If the file is not found, an error will be issued and compilation shall stop.

Purpose: Text from the include file **filename.h** is inserted at the point of the source where this directive appears, at compile time.

Examples:

```
#include <system.h>
```

#define

Syntax: *#define* id statement

or

#define id(a, b...) statement

Elements: **id** is any valid preprocessor identifier.

statement is any valid text.

a, b and so on are local preprocessor identifiers, that in the given form model a function's formal parameters, separated by commas.

Purpose: Both forms produce a basic string replacement of **id** with the given **text**. Replacement will take place from the point where the *#define* statement appears in the program, and below.

The second form represents a preprocessor pseudo-function. The local identifiers are positionally matched up with the original text, and are replaced with the text passed to the macro wherever it is used.

Examples:

```
#define LEN 16
#define LOWNIBBLE(x) ((x) & 0x0F)
...

a = 69;

le = a + LEN;          // becomes le = a + 16;

b = LOWNIBBLE(a);      // same as b = a & 0x0F;
```

#undef

Syntax: *#undef* `id`

Elements: **id** is any valid preprocessor identifier previously defined via *#define*.

Purpose: Starting with the line where this directive appears, **id** will no more have meaning for the preprocessor, i.e. a subsequent *#ifdef* ***id*** shall evaluate to logical FALSE.

Please note that **id** can then be reused and assigned a different value.

Examples:

```
#define LEN 16
#define LOWNIBBLE(x) ((x) & 0x0F)
...
a = 69;
le = a + LEN;          // becomes le = a + 16;

#undef LEN             // LEN is not recognized anymore by pp
...

#define LEN 24         /* This is now valid and does not cause
“double definition attempt” errors. */
```

#if, #else, #endif

Syntax:

```
#if expr
    code
#else
    code
#endif
```

Elements: **expr** is any valid expression using constants, standard operators and preprocessor identifiers.
code is one or more valid C source code line.

Purpose: The preprocessor evaluates the constant expression **expr** and, if it is non-zero, will process the lines up to the optional *#else* or the *#endif*. Otherwise the optional *#else* branch code will be processed, if present.

The latter two preprocessor directives are also used with specialized forms of the *#if* directive (see ***#ifdef***, ***#ifndef***).

NOTE: **expr** cannot contain C variables ! Only constant expressions and operators can be used.

Examples:

```
// Conditionally initialize a RAM variable

#if STARTDELAY > 20
    slow = 1;
#else
    slow = 0;
#endif
```

#ifdef

Syntax:

```
#ifdef id
    code
#endif
```

Elements: **id** is any valid preprocessor identifier.

code is one or more lines of valid C source code.

Purpose: When the preprocessor encounters this directive, it evaluates whether the identifier **id** is in its symbol table (eg previously specified within a *#define* statement).
In case **id** is defined, the lines of **code** between *#ifdef* and *#endif* (or an optional *#else*, if present) will be processed.

In the opposite case, **code** statements between *#ifdef* and *#endif* will be ignored by the compiler.

NOTE: **id** can not be a C variable ! Only preprocessor identifiers created via *#define* can be used.

Examples:

```
#define DEBUG
...

#ifdef DEBUG
    printf("Reached test point #1");
#endif
```

#ifndef

Syntax:

```
#ifndef id
    code
#endif
```

Elements: **id** is any valid preprocessor identifier.

code is one or more lines of valid C source code.

Purpose: When the preprocessor encounters this directive, it evaluates whether the identifier **id** is in its symbol table (eg previously specified within a *#define* statement).

In case **id** is **not** defined, the lines of **code** between *#ifndef* and *#endif* (or an optional *#else*, if present) will be processed.

In the opposite case, **code** statements between *#ifndef* and *#endif* will be ignored by the compiler.

NOTE: **id** can not be a C variable ! Only preprocessor identifiers created via *#define* can be used.

Examples:

```
#ifndef DEBUG
    printf("Debug disabled !");
#else
    printf("Reached test point #1");
#endif
```

#error

Syntax: *#error* **text**

Elements: **text** is any valid text.

Purpose: When the preprocessor encounters this directive, it stops compilation and issues and error. The user supplied **text** is printed as an informational message.

This directive is useful when coupled with the expression checking features of the preprocessor, to validate the coherence of configuration choices and defines made elsewhere in the sources and include files (or on the command line).

Examples:

```
#ifndef PWM_DEFAULT
  #error "MUST define a default value for speed !"
#endif
```

#warning

Syntax: *#warning* **text**

Elements: **text** is any valid text.

Purpose: When the preprocessor encounters this directive, it forces the compiler to issue a warning. The user supplied **text** is printed as an informational message.

This directive is useful when coupled with the expression checking features of the preprocessor, to validate the coherence of configuration choices and defines made elsewhere in the sources and include files (or on the command line).

Examples:

```
#ifndef NODEADDR
  #warning "ADDR not defined, will enter dynamic mode."
#endif
```

Pragma directives

Specific BoostC preprocessor directives all follow the ANSI keyword *#pragma*, so to avoid potential conflicts when porting code to other compilers and/or with advanced source analysis tools (lint, static checkers, code formatters, flow analyzers and so on).

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, ... "label"**

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

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

"label" is an optional text label used to reference data.

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"
//Set PIC16 configuration word
#pragma DATA 0x2007, _HS_OSC & _WDT_OFF & _LVP_OFF
//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:

```
//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:

```
//Use aggressive optimization for function 'foo'
#pragma OPTIMIZE "a"

void foo()
{
  ...
}
```

This page is intentionally left blank.

C language

This section of the manual contains a condensed list of BoostC C compiler features. It is in no way intended to replace a complete C language manual or ANSI/ISO specification. It is targeted, instead, at the already expert C programmer that needs a quick reference of BoostC and its peculiarities due to the specific PIC target platform.

Program structure

Every source file should include the general system header file, that in turn includes target specific header (containing register mapped variables specific for this target), some internal functions prototypes needed for code generation and string manipulation function prototypes:

```
#include <system.h>
```

Data types

Base data types

Size	Type
1 bit	bit, bool
8 bits	char, unsigned char, signed char
16 bits	short, unsigned short, signed short
16 bits	int, unsigned int, signed int
32 bits	long, unsigned long, signed long

The difference between bit and bool data types is in the way how an expression (longer than 1 bit) is assigned to a bit or bool operands.

- **bit** operands receive the least significant bit of the right side expression;
- **bool** operands receive the value of the right side expression casted to bool.

For example:

```
bool a;
bit  b;
char x;
...
a = x & 2; // 'a' will be 'true' if the bit #1 in 'x' is set
           // and 'false' otherwise

b = x & 2; // 'b' will always be false, because bit #0
           // (the least significant bit) in the expression
           // result is zero - regardless of the value of 'x'
```

Structures and unions

Both struct and union keywords are supported.

Typedef

New names for data types can be defined using typedef operation:

```
typedef unsigned char uchar;
```

Enum

Enumerated data types are an handy type of automatically defined constant series. The declaration assigns a value of zero to the first symbolic constant of the list, and the assigns subsequent values (automatically incremented) to the following constants.

The user can, as well, arbitrarily assign numerical (signed) values at the beginning as well as in the middle of the series. Values following an explicit assignment use that value as a base and keep on incrementing from that point.

The data type for an enum type or typedef variable is, as per ANSI definition, *the smaller type that can contain the absolute maximum value of the constant series*.

```
enum ETypes { E_NONE=0, E_RED, E_GREEN, E_BLUE };

// Same as :
// #define E_NONE  0
// #define E_RED   1
// #define E_GREEN 2
// #define E_BLUE  3
```

Code size vs Data Types

Be sure to always use the smallest data types possible. The rule is simple: the bigger data types are used, the bigger code will be generated.
Thus, always follow these rules of thumb:

- Use char (8-bit or **byte**) as the default, everywhere;
- Use short or int (16-bit or **word**) for common arithmetic, counters and to hold ADC conversion results on advanced cores (with 10-bit or more internal ADC).
- Only as a as last resort, and only where absolutely necessary, use long (32-bit, **dword**) variables.

Another rule that also affects the size of produced code, though in a much smaller degree, is about sign.

Use **unsigned data types** wherever you can, and signed only when necessary. Unsigned math always generates smaller (and typically faster) code than signed.

Rom

Strings or arrays of data can be placed into program memory.

Such variables are declared using regular data types and rom storage specifier.

Such rom variables must be initialized within declaration:

```
rom char *text = "Test string";    // text with trailing zero
rom char *ptr  = "\0x64\11\12";    // 4 bytes: 0x64, 0x0B, 0x0C, 0x00
rom char *data = { 0x64, 11, 12 }; // 3 data bytes: 0x64, 0x0B, 0x0C
```

Please keep in mind that the rom storage specifier has several limitations:

- rom can be used with char data types only;
- there is no implicit cast between rom and regular data types. Though BoostC will not generate an explicit error for such a cast, it is expected that the operand should be casted back to its original type.
  If this is not done, the resulting code will behave unpredictably.
- a rom pointer is internally limited to 8-bits: the constant array size is thus limited to 256 elements. This is coherent with smaller cores constraints;
- access to rom elements has to be done exclusively through the **[]** operators and they **cannot be referenced with substring pointer initialized at runtime.** Please keep in mind that rom variables must always and exclusively be initialized within declaration;

Example of wrong referencing with a runtime initialized pointer:

```
/*
Part of the following code is WRONG and will FAIL because BoostC cannot
dynamically create the pointer to mystr[OFFSET] when the mystr array is
located in ROM.
*/

rom char *mystr = "Str_one \0 Str_two \0";

/* WRONG: a rom pointer must be initialized in declaration */
rom char *substr;

    substr = &mystr[OFFSET];        //** WRONG **
    cc = substr[0];                 //** WRONG **

    cc = mystr[OFFSET];             // Correct
```

Volatile

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:

```
volatile bit pinB1@0x6.1; //declare bit variable mapped to pin 1, port B
```

Currently compiler generates different code only for expressions that assign values to volatile bit variables. Also volatile variables are not checked for being initialized.

Static

Both global and local variables can be declared as static. This limits their scope to the current module.

Constants

Constants can be expressed in binary, octal, decimal and hexadecimal forms:

0bXXXX or XXXXb	binary number, where X is either 1 or 0
0XXXX	octal number, where X is a number between 0 and 7
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

Strings

Besides regular characters, strings can include escape sequences:

\nn	ASCII character, value nn is decimal
\xnn	ASCII character, value nn is hexadecimal
\a	ASCII character 0x07 (ALERT)
\b	ASCII character 0x08 (Backspace)
\t	ASCII character 0x09 (Horizontal Tab)
\r	ASCII character 0x0A (LF, Line Feed)
\v	ASCII character 0x0B (Vertical Tab)
\f	ASCII character 0x0C (Form Feed)
\n	ASCII character 0x0D (CR, Carriage Return)
\\	ASCII character 0x5C (the «\» character)
\'	ASCII character 0x27 (the «'» character)
\"	ASCII character 0x22 (the «"» character)
\?	ASCII character 0x3F (the «?» character)

Variables

Variables can be declared and used in the standard ANSI C way.
The linker will place variables at specific addresses. BoostLink analyzes the call and scope trees, so that it can re-use the same pool of RAM memory locations for variables that don't collide with each other, being used disjointly by unrelated routines active at different times.

This is a very effective way to minimize data memory usage.

Register mapped variables

Variables can be forced to be placed at certain addresses. Syntax is the same as in the legacy C2C compiler:

```
char var@<addr>;
```

where <addr> is an hex or decimal address.
This technique is used to access target specific registers from code.
Please note that system header files already contain all target specific registers, so there is no need to define them again in the user's code.

Bit variables can also have fixed addresses. Their address includes bit position and can be made in 2 forms:

```
bit b;          //variable will be placed by linker at arbitrary position
bit b1@0x40.1; // dotted: access bit 1 of register 0x40
bit b2@0x202;  // bit offset: access bit 2 of register 0x40 (0x40*8 + 2)
```

Bit access

Besides 'bit' variables, individual bits of every variable can be accessed using the '.' operator:

```
char var;
var.2 = 1; //set bit 2 of variable 'var'
```

Arrays

Arrays can have any number of dimensions. The only constraint is that an array must fit into a single RAM bank.

Pointers

Pointers can be used in the standard general way, the only exception being variables declared with the **rom** storage specifiers, that can only be accessed through the [] operators.

Strings as function arguments

If a function has one or more char* arguments, it can be called with a constant string passed as an argument.
The compiler will reuse the same RAM memory allocated for such arguments when several similar calls are made within same code block.
For example, the code below will use the same memory block to temporarily store the strings "Date" and "Time":

```
...
foo( "Date" );
foo( "Time" );
...
```

Operators

If an operation result is not explicitly casted, it is promoted by default to 16 bit precision. For example, given the following expression:

```
long l = a * 100 / b; //'a' and 'b' are 16 bit long variables
```

the result of the multiplication will be stored in a 16-bit long (**word**) temporary variable, that will be then divided by **b.** This 16-bit long result will eventually be stored in **l**. This is the ANSI 'C' standard behavior.

This behavior can be changed using the -Op compiler command line option or a local *#pragma OPTIMIZE* directive.
When this optimization is applied to the given expression, the multiplication result will be promoted to 32-bit long (**dword**) temporary variable, that will then be divided by **b:** the result, that is now 32 bit long, will eventually be stored in **l**.

Arithmetic

+ - * / % ++ --

Arithmetic Operator Examples

```
//  +   is a binary operator. It is used to add or produce
//      the arithmetic sum of two operands.

// Example:

          c = a + b;
//    or
          c = 5 + 7;   // After the operation c = 12


//  -   is a binary operator. It is used to subtract or produce
//      the difference of two operands. In other words, the second
//      operand is subtracted from the first operand.

// Example:

          c = a - b;
//    or
          c = 18 - 12;   // After the operation c = 6
```

```
//  *   is a binary operator. It is used to multiply or produce
//      the arithmetic product of two operands.

// Example:

          c = a * b;
//    or
          c = 5 * 7;   // After the operation c = 35
```

```
//  /   is a binary operator. It is used to divide or produce
//      the quotient of two operands. In other words, the first
//      operand is divided by the second operand.

// Example:

          c = a / b;
//    or
          c = 24 / 8;   // After the operation c = 3
```

```
//  %   is a binary operator. It is used to produce the modulus
//      or remainder when two operands are divided.

// Examples:

     c = a % b;
// or
     c = 25 % 8;   // After the operation  c = 1

     C = 17 % 3;   // 17 % 3 = 5 with a remainder of 2
                   // After the operation  c = 2

     c = 17 % 4;   // 17 % 4 = 4 with a remainder of 1
                   // After the operation  c = 1
```

```
//  ++   is a unary operator. It is used for pre-incrementing
//       or post-incrementing an operand.

// Examples:

     x = 10;
     c = x++;   // Post-increment.
                // After the operation x = 11, c = 10

     x = 10;
     c = ++x;   // Pre-increment.
                // After the operation x = 11, c = 11
```

```
//  --   is a unary operator. It is used for pre-decrementing
//       or post-decrementing an operand.

// Examples:

     x = 10;
     c = x--;   // Post-decrement.
                // After the operation x = 9, c = 10

     x = 10;
     c = --x;   // Pre-decrement.
                // After the operation x = 9, c = 9
```

Assignment

```
=   +=   -=   *=   /=   %=   &=   |=   ^=   <<=   >>=
```

Assignment Operator Examples

```
//  =   is the ASSIGN operator. The value of the variable or
//      expression on the right side of the equal is assigned
//      to the variable on the left side of the equal.

// Examples:

     x = 3;   // Whatever was in the variable x
              // has been replaced with 3.

     x = 2 + 4;   // Whatever was in the variable x
                  // has been replaced with 6.

     c = x + y;   // If x has a value of 12 and y has a value of 16.
                  // Whatever was in the variable c will
                  // be replaced with 28.
```

```
//  +=   is the combined ADD and ASSIGN operator. The variable
//       on the left side of the operator will be added to the
//       variable or expression on the right side. The result is
//       then assigned to the variable on the left side of the
//       operator.

// Examples:

     x += 2;   // If x has an initial value of 14. After
               // the operation x will be 16.

     c += x + y;   // If c has an initial value of 10 and x has
                   // the value 12 and y has the value 16.
                   // After the operation c will be 38.
```

```
//  -=   is the combined SUBTRACT and ASSIGN operator. The variable
//       or expression on the right side of the operator will be
//       subtracted from the variable on the left side. The result
//       is then assigned to the variable on the left side of the
//       operator.

// Examples:

     x -= 2;   // If x has an initial value of 14. After
               // the operation x will be 12.

     c -= x + y;   // If c has an initial value of 38 and x has
                   // the value 12 and y has the value 16.
                   // After the operation c will be 10.
```

```
//  *=   is the combined MULTIPLY and ASSIGN operator. The variable
//       on the left side of the operator will be multiplied by the
//       variable or expression on the right side. The result is
//       then assigned to the variable on the left side of the
//       operator.

// Examples:

     x *= 2;   // If x has an initial value of 14. After
               // the operation x will be 28.

     c *= x + y;   // If c has an initial value of 10 and x has
                   // the value 12 and y has the value 16.
                   // After the operation c will be 280.
```

```
//  /=   is the combined DIVIDE and ASSIGN operator. The variable
//       on the left side of the operator will be divided by the
//       variable or expression on the right side. The result
//       is then assigned to the variable on the left side of
//       the operator.

// Examples:

     x /= 2;    // If x has an initial value of 14. After
                // the operation x will be 7.

     c /= x + y;    // If c has an initial value of 280 and x has
                    // the value 12 and y has the value 16.
                    // After the operation c will be 10.
```

```
//  %=   is the combined MODULUS and ASSIGN operator. The variable
//       on the left side of the operator will be divided by the
//       variable or expression on the right side. The result,
//       which is a remainder only, is then assigned to the
//       variable on the left side of the operator.

// Examples:

     x %= 2;    // If x has an initial value of 15. After
                // the operation x will be 1.

     y %= 7;    // If y has an initial value of 17. After
                // the operation y will be 3.

     c %= x + y;    // If c has an initial value of 19 and x has
                    // the value 4 and y has the value 3.
                    // After the operation c will be 5.
```

```
//  &=   is the combined BITWISE-AND and ASSIGN operator. The variable
//       on the left side of the operator will be ANDed on a bit-by-bit
//       basis with the variable or constant on the right side. The
//       result is then assigned to the variable on the left side of
//       the operator.

// Examples:

     x &= y;    // If x has an initial value of 14 and y has the
                // value 5. After the operation x will be 4.

     c &= 0x07;    // If c has an initial value of 0x0E. After
                   // the operation c will be 0x06.

     y &= 0b11110001;    // If y has an initial value of 0b10001111.
                         // After the operation y will
                         // be 0b10000001.
```

```
//  |=   is the combined BITWISE-OR and ASSIGN operator. The variable
//       on the left side of the operator will be ORed on a bit-by-bit
//       basis with the variable or constant on the right side. The
//       result is then assigned to the variable on the left side of
//       the operator.

// Examples:

     x |= y;   // If x has an initial value of 14 and y has the
               // value 5. After the operation x will be 15.

     c |= 0x07;   // If c has an initial value of 0x0E. After
                  // the operation c will be 0x0F.

     y |= 0b11110000;   // If y has an initial value of 0b10001110.
                        // After the operation y will
                        // be 0b11111110.
```

```
//  ^=   is the combined BITWISE-XOR and ASSIGN operator. The variable
//       on the left side of the operator will be XORed on a bit-by-bit
//       basis with the variable or constant on the right side. The
//       result is then assigned to the variable on the left side of
//       the operator.

// Examples:

     x ^= y;   // If x has an initial value of 14 and y has the
               // value 5. After the operation x will be 11.

     c ^= 0x07;   // If c has a value of 0x0E. After the
                  // operation c will be 0x09.

     y ^= 0b11111000;   // If y has an initial value of 0b00011110.
                        // After the operation y will
                        // be 0b11100110.
```

```
//  <<=   is the combined SHIFT-LEFT and ASSIGN operator. The variable
//        on the left side of the operator will be shifted left by the
//        number of places indicated by the variable or constant on
//        the right. The result is then assigned to the variable on
//        the left side of the operator.

// Examples:

     x <<= y;   // If x has an initial value of 14 and y has the
                // value 2. After the operation x will be 56.

     c <<= 0x01;   // If c has an initial value of 0x0E. After the
                   // operation c will be 0x1C.

     y <<= 0b00000010;   // If y has an initial value of 0b00011110.
                         // After the operation y will
                         // be 0b01111000.
```

```
//  >>=  is the combined SHIFT-RIGHT and ASSIGN operator. The
//       variable on the left side of the operator will be shifted
//       right by the number of places indicated by the variable
//       or constant on the right. The result is then assigned to
//       the variable on the left side of the operator.

// Examples:

     x >>= y;   // If x has an initial value of 14 and y has the
                // value 2. After the operation x will be 3.

     c >>= 0x01;   // If c has an initial value of 0x0E. After the
                   // operation c will be 0x07.

     y >>= 0b00000010;   // If y has an initial value of 0b00011110.
                         // After the operation y will
                         // be 0b00000111.
```

Comparison

```
     ==  !=  <  <=  >  >=
```

Comparison Operator Examples

```
//  ==  is a binary operator. It is used to see if one operand
//      IS equal to another operand.

// Example1:

     if( x == y )    // If c has an initial value of 0, and
     {               // x has the value 8 and y has the value 8.
        c = x + y;   // The final value for c will be 16.
     }

// Example2:

     if( x == y )    // If c has an initial value of 0, and
     {               // x has the value 8 and y has the value 5.
        c = x + y;   // The final value for c will be 0.
     }
```

```
//  !=  is a binary operator. It is used to see if one operand
//      is NOT equal to another operand.

// Example1:

     if( x != y )    // If c has an initial value of 0, and
     {               // x has the value 8 and y has the value 5.
        c = x * y;   // The final value for c will be 40.
     }

// Example2:

     if( x != y )    // If c has an initial value of 0, and
     {               // x has the value 8 and y has the value 8.
        c = x * y;   // The final value for c will be 0.
     }
```

```
//  <   is a binary operator. It is used to see if one operand
//      is LESS than another operand.

// Example1:

     if( x < y )      // If c has an initial value of 0, and
     {                // x has the value 40 and y has the value 65.
        c = y - x;    // The final value for c will be 25.
     }

// Example2:

     if( x < y )      // If c has an initial value of 0, and
     {                // x has the value 65 and y has the value 40.
        c = y - x;    // The final value for c will be 0.
     }
```

```
//  <=   is a binary operator. It is used to see if one operand
//       is LESS than or EQUAL to another operand.

// Example1:

     if( x <= y )   // If x has a value of 22 and y has a value of 33.
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// In this example the LED will be turned ON.

// Example2:

     if( x <= y )   // If x has a value of 15 and y has a value of 8.
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// In this example the LED will be turned OFF.

// Example3:

     if( x <= y )   // If x has a value of 46 and y has a value of 46.
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// In this example the LED will be turned ON.
```

```
//  >   is a binary operator. It is used to see if one operand
//      is GREATER than another operand.

// Example1:

     if( x > y )      // If c has an initial value of 0, and
     {                // x has the value 28 and y has the value 14.
        c = x / y;    // The final value for c will be 2.
     }

// Example2:

     if( x > y )      // If c has an initial value of 0, and
     {                // x has the value 14 and y has the value 28.
        c = x / y;    // The final value for c will be 0.
     }
```

```
//  >=   is a binary operator. It is used to see if one operand
//       is GREATER than or EQUAL to another operand.

// Example1:

     if( x >= y )   // If x has a value of 25 and y has a value of 10.
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// In this example the LED will be turned ON.

// Example2:

     if( x >= y )   // If x has a value of 8 and y has a value of 15.
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// In this example the LED will be turned OFF.

// Example3:

     if( x >= y )   // If x has a value of 34 and y has a value of 34.
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// In this example the LED will be turned ON.
```

Logical

&& || !

Logical Operator Examples

```
//  &&  is a binary operator. It is used to determine if both operands
//      are true. The operands are expressions that evaluate to true
//      or false.

// Example1:

     if( ( temp > 50 ) && ( temp < 100 ) )
        clear_bit( PORTA, ALARM_bit );    // Turn alarm OFF
     else
        set_bit( PORTA, ALARM_bit ); // Turn alarm ON

// If temp has a value of 70 the alarm will be turned OFF.

// Example2:

     if( ( temp > 50 ) && ( temp < 100 ) )
        clear_bit( PORTA, ALARM_bit );    // Turn alarm OFF
     else
        set_bit( PORTA, ALARM_bit ); // Turn alarm ON

// If temp has a value of 105 the alarm will be turned ON.

// Example3:

     if( ( temp > 50 ) && ( temp < 100 ) )
        clear_bit( PORTA, ALARM_bit );    // Turn alarm OFF
     else
        set_bit( PORTA, ALARM_bit ); // Turn alarm ON

// If temp has a value of 25 the alarm will be turned ON.
```

```
//  ||   is a binary operator. It is used to determine if either
//       operand is true. The operands are expressions that evaluate
//       to true or false.

// Example1:

     if( ( volt > 7 ) || ( volt < 5 ) )
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// If volt has a value of 8 the LED will be turned ON.

// Example2:

     if( ( volt > 7 ) || ( volt < 5 ) )
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// If volt has a value of 6 the LED will be turned OFF.

// Example3:

     if( ( volt > 7 ) || ( volt < 5 ) )
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// If volt has a value of 4 the LED will be turned ON.
```

```
//  !   is a unary operator. It is used to complement an evaluated
//      operand. The operand is an expression that evaluates to
//      true or false.

// Example1:

     if( !( pressure > 120 ) )
        clear_bit( PORTA, ALARM_bit );    // Turn alarm OFF
     else
        set_bit( PORTA, ALARM_bit ); // Turn alarm ON

// If pressure has a value of 75 the alarm will be turned OFF.

// Example2:

     if( !( pressure > 120 ) )
        clear_bit( PORTA, ALARM_bit );    // Turn alarm OFF
     else
        set_bit( PORTA, ALARM_bit ); // Turn alarm ON

// If pressure has a value of 125 the alarm will be turned ON.
```

Bitwise

& | ^ ~ << >>

Bitwise Operator Examples

```
//  &  is a binary operator. It is used to produce the logical product
//     of two operands. The individual bits of two operands are ANDed
//     together to produce the final results.

// Examples:

     c = x & y;   // If x has a value of 14 and y has a value
                  // of 5. After the operation c will be 4.

     x = y & 0x07;   // If y has a value of 0x0E. After the
                     // operation x will be 0x06.

     x = y & 0b11110001;   // If y has a value of 0b10001111.
                           // After the operation x will
                           // be 0b10000001.
```

```
//  |  is a binary operator. It is used to produce the logical sum of
//     two operands. The individual bits of two operands are ORed
//     together to produce the final results.

// Examples:

     c = x | y;   // If x has a value of 14 and y has a value
                  // of 5. After the operation c will be 15.

     x = y | 0x07;   // If y has a value of 0x0E. After the
                     // operation x will be 0x0F.

     x = y | 0b11110000;   // If y has a value of 0b10001110.
                           // After the operation x will
                           // be 0b11111110.
```

```
//  ^  is a binary operator. It is used to produce the logical
//     difference of two operands. The individual bits of two operands
//     are XORed together to produce the final results.

// Examples:

     x = y ^ 0x07;   // If y has a value of 0x0E. After the
                     // operation x will be 0x09.

     x = y ^ 0b11111000;   // If y has a value of 0b00011110.
                           // After the operation x will
                           // be 0b11100110.
```

```
//  ~   is a unary operator. It is used to produce the complement of an
//      operand. The individual bits of the operand are complemented.
//      The ones become zeros and the zeros become ones.

// Examples:

     x = ~y;    // If y has a value of 0x0E. After the
                // operation x will be 0xF1.

     x = ~0b01010111;    // After the operation x will
                         // be 0b10101000.
```

```
//  <<   is a binary operator. The operand on the left side of the
//       operator will be shifted left by the number of places
//       indicated by the operand on the right.

// Examples:

     c = x << y;    // If x has a value of 14 and y has a value
                    // of 2. After the operation c will be 56.

     x = y << 0x01;    // If y has a value of 0x0E. After the
                       // operation x will be 0x1C.

     x = y << 0b00000010;    // If y has a value of 0b00011110.
                             // After the operation x will
                             // be 0b01111000.
```

```
//  >>   is a binary operator. The operand on the left side of the
//       operator will be shifted right by the number of places
//       indicated by the operand on the right.

// Examples:

     c = x >> y;    // If x has a value of 14 and y has a value of 2.
                    // After the operation c will be 3.

     x = y >> 0x01;    // If y has a value of 0x0E. After the operation
                       // x will be 0x07.

     x = y >> 0b00000010;    // If y has a value of 0b00011110.
                             // After the operation x will
                             // be 0b00000111.
```

Conditionals

- if / else statement
- switch statement
- ? : ternary operator

Conditional Examples

```
//  if / else  is a two-way decision making statement. If the
//             expression evaluates to true, the first statement
//             will be executed. If it evaluates to false the
//             second statement will be executed.

// Example1:

     if( x > y )   // If x has a value of 25 and y has a value of 10.
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// In this example the LED will be turned ON.

// Example2:

     if( x > y )   // If x has a value of 8 and y has a value of 15.
        set_bit( PORTA, LED_bit );        // Turn LED ON
     else
        clear_bit( PORTA, LED_bit ); // Turn LED OFF

// In this example the LED will be turned OFF.
```

```
//  switch  is a multi-way decision making statement. The variable is
//          compared with the different cases. The case that matches
//          will have its statements executed.

// Example1:

     switch( weight )
     {
        case 5:
           set_bit( PORTA, red_LED );        // Turn red LED ON
           clear_bit( PORTA, green_LED );    // Turn green LED OFF
           break;
        case 10:
           set_bit( PORTA, green_LED );      // Turn green LED ON
           clear_bit( PORTA, red_LED );      // Turn red LED OFF
           break;
        default:
           clear_bit( PORTA, green_LED );    // Turn green LED OFF
           clear_bit( PORTA, red_LED );      // Turn red LED OFF
     }
// If the 'weight' variable has a value of 5 the red LED will
// be turned ON and the green LED will be turned OFF.

// Example2:

     switch( weight )
     {
        case 5:
           set_bit( PORTA, red_LED );        // Turn red LED ON
           clear_bit( PORTA, green_LED );    // Turn green LED OFF
           break;
        case 10:
           set_bit( PORTA, green_LED );      // Turn green LED ON
           clear_bit( PORTA, red_LED );      // Turn red LED OFF
           break;
        default:
           clear_bit( PORTA, green_LED );    // Turn green LED OFF
           clear_bit( PORTA, red_LED );      // Turn red LED OFF
     }
// If the 'weight' variable has a value of 10 the green LED will
// be turned ON and the red LED will be turned OFF.

// Example3:

     switch( weight )
     {
        case 5:
           set_bit( PORTA, red_LED );        // Turn red LED ON
           clear_bit( PORTA, green_LED );    // Turn green LED OFF
           break;
        case 10:
           set_bit( PORTA, green_LED );      // Turn green LED ON
           clear_bit( PORTA, red_LED );      // Turn red LED OFF
           break;
        default:
           clear_bit( PORTA, green_LED );    // Turn green LED OFF
           clear_bit( PORTA, red_LED );      // Turn red LED OFF
     }

// If the 'weight' variable has any value other than 5 or 10,
// both the green and red LEDs will be turned OFF.
```

```
//  ? :  is an if/else operator. This operator can be used inside an
//       expression to determine if a part of it is true or false.

// Example1:              turn the LED ON             turn the LED OFF

     (volts > 5) ? set_bit(PORTA, LED_bit) : clear_bit(PORTA, LED_bit);

// If 'volts' has a value of 7 turn the LED ON.


// Example2:              turn the LED ON             turn the LED OFF

     (volts > 5) ? set_bit(PORTA, LED_bit) : clear_bit(PORTA, LED_bit);

// If 'volts' has a value of 3 turn the LED OFF.
```

Program Flow

while
do / while
for
break
continue

Program Flow Examples

```
//  while  is a loop control construct. It controls the execution of
//         a block of statements for as long as an expression evaluates
//         to true. The expression is evaluated first and if true,
//         executes the block. If it evaluates to false, stop the
//         execution.

// Example1:

     while( number > 0 )
     {
        factorial *= number;    // 'factorial' is initialized to 1
        --number;               //  before entering the loop.
     }

// If 'number' has a value of 3 'factorial' will become 6.
// factorial = 3 x 2 x 1 ;

// Example2:

     while( number > 0 )
     {
        factorial *= number;    // 'factorial' is initialized to 1
        --number;               //  before entering the loop.
     }

// If 'number' has a value of 0, 'factorial' will stay
//  equal to 1 because the loop was never entered.
```

```
//  do / while  is a loop control construct. It controls the execution
//              of a block of statements for as long as an expression
//              evaluates to true. The block is executed at least once
//              before the expression is evaluated. If it evaluates to
//              false, stop the execution. If it evaluates to true,
//              continue the execution.

// Example1:

     do
     {
        factorial *= number;    // 'factorial' is initialized to 1
        --number;               //  before entering the loop.

     } while( number > 0 );

// If 'number' has a value of 4 'factorial' will become 24.
// factorial = 4 x 3 x 2 x 1 ;

// Example2:

     do
     {
        factorial *= number;    // 'factorial' is initialized to 1
        --number;               //   before entering the loop.

     } while( number > 0 );

// If 'number' has a value of 0, 'factorial' will become 0 because
//  the loop was entered, before the expression was evaluated.
```

```
//  for  is a loop control construct. It controls the number of times
//       a block of statements is executed. The construct has an
//       initial value, a final value, and a loop-count value that is
//       incremented each time after the block is executed.

// Example1:

     for( volts = 0; volts < 7; volts++ )
     {
        sum += volts;      // 'sum' is initialized to 0
     }                     //   before entering the loop.

// Upon exiting the loop 'sum' will have a value of 21.
```

```
//  break  is an option that can be used to exit out of a for-loop,
//         based upon the evaluation of an expression.

// Example1:

     for( volts = 0; volts < 7; volts++ )
     {
        if( volts == 5 )
           break;

        sum += volts;      // 'sum' is initialized to 0
     }                     //   before entering the loop.

// Upon exiting the loop 'sum' will have a value of only 10.
```

```
//  continue  is an option used to redirect a for-loop based upon the
//            evaluation of an expression. If the expression evaluates
//            to true, the block of statements will not be executed.

// Example1:

     for( volts = 0; volts < 7; volts++ )
     {
        if( volts == 5 )
           continue;

        sum += volts;      // 'sum' is initialized to 0
     }                     //   before entering the loop.

// Upon exiting the loop 'sum' will have a value of only 16.
// 'sum' will only have the values 0, 1, 2, 3, 4, & 6 added together.
```

Goto

In the vast majority of programming books, the usage of 'goto' is heavily deprecated. This is true for BoostC and PIC C coding as well: it should normally be avoided.

There are, anyway, some very specific circumstances where it may still be useful: to optimize early exit cases within complex nested control structures or to simplify local error handling (it can somehow mimic try/catch exception handling syntax).

```
while( ... )
{
  while(...)
  {
    while(...)
    {
      goto exit;
    }
  }
}
exit:
```

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.
The asm and _asm operators differ in that the compiler will not apply any optimization for the assembly code used with the asm operator (will be assembled **as is**), while with _asm all usual optimizations will be applied, including elimination of dead code.

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.

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

User Data

User data can be placed at the current location using the 'data' assembly instruction followed by comma separated numbers or strings.
Example:

```
// Code below will place bytes 10,11,116,101,115,116,0
// at current code location
asm data 0xA, 0xB, "test"
```

Functions

Inline functions

Functions declared as inline are repeatedly embedded into the code for each occurrence. When a function is defined as inline, its body must be defined before it gets called for the first time.

Though any function can be declared as inline, procedures (functions with no return value and a possibly empty argument list) are best suited to be used as inline.

An heavy usage of inline functions obviously augments code size.

Special functions

```
void main(void)
```

Program entry point. This function is mandatory for every C program.

```
void interrupt(void)
```

Interrupt handler function. Is linked to high priority interrupts for PIC18 parts.

```
void interrupt_low(void)
```

Low priority interrupt handler, can be used only on the PIC18 family.

General functions and interrupts

Standard user functions are not *thread-safe*: their local variables are **not** saved when function execution gets interrupted by an interrupt. This can lead to very hard to trace errors.

To prevent this pitfall, the linker does not allow to call a given function from both main() and interrupt threads.

If you really need to use same function in both threads, you need to duplicate its code and assign a different name to the second copy.

```
//This function gets called from main thread
void foo()
{
  ...
}

//Copy of 'foo' that will be called from interrupt thread
void foo_interrupt()
{
  ...
}

//Interrupt thread
void interrupt( void )
{
  ...
  foo_interrupt();
  ...
}

//Main thread
void main( void )
{
  ...
  foo();
  ...
}
```

Dynamic memory management

Dynamic memory management is used to dynamically create and destroy objects at run time.
For example, this functionality may be needed when a program needs to keep several data packets. Memory for this packets can also be allocated at compile time, but this way the memory may not be available for other variables even if it's not used.
The solution is to use dynamic memory allocation. Objects to store data are created as soon as they are needed and destroyed after data gets processed.
This way all available target data memory is used most efficiently.

The amount of possible objects that can be allocated depends on the specific PIC part at hand, and on the application.

When the application is built, the linker uses RAM memory left after allocation of global and local variables as a heap. When some memory gets allocated at run time by the 'alloc' call, it gets allocated from this heap. The bigger the heap, the more run time objects can exist at any given time.

```
void* alloc(unsigned char size)
```

Dynamically allocate memory 'size' bytes long. Max size is 127 bytes. Returns NULL if memory can't be allocated.

```
void free(void *ptr)
```

Free memory previously allocated by 'alloc'. Passing any other pointer will lead to unpredictable results.

C language superset

The BoostC compiler has some advanced features "borrowed" from C++ language. These features allow to develop more flexible and powerful code, but their use is merely optional.

References as function arguments

Function arguments can be references to other variables.
When such argument changes inside a function the original variable used in function call changes too.
This is a very powerful way to alter the data flow without blowing up the generated code:

```
void foo(char &n)          //'n' is a reference
{
  n = 100;
}

void main(void)
{
  char a = 0;
  foo( a );                //upon return 'a' will have value of 100
  ...
}
```

Function overloading

There can be more than one function in the same application having a given name. Such functions must anyway differ by the number and type of their arguments:

```
void foo( void )                //'foo' number 1
{
...
}

void foo( char *ptr )           //'foo' number 2
{
...
}

void foo( char a, char b )      //'foo' number 3
{
...
}

void main( void )
{
  foo();              //'foo' number 1 gets called
  foo( "test" );      //'foo' number 2 gets called
  foo( 10, 20 );      //'foo' number 3 gets called
  ...
}
```

The compiler will generate internal references to the functions so that no ambiguity is possible (*name mangling*), and will select which function will be invoked for each call analyzing how many parameters are passed, as well as their type.

Function templates

Functions can be declared and defined using data type placeholders.
This feature allows writing very general code (for example, linked lists handling) that is not tied to a particular data type and, what may be more important, allows the user to create template libraries contained in header files:

```
template <class T>
void foo( T *t )
{
...
}

void main( void )
{
  short s;
  foo<char>( "test" );     //'foo( char* )' gets called
  foo<short>( &s );        //'foo( short* )' gets called
  ...
}
```

Parametric timing functions

Most of software based timing functions are strictly dependent on clock speed.
As this parameter is usually well known at linking time, depending only on hardware design and implementation, such functions can be dynamically generated, once the clock frequency is correctly assigned with the CLOCK_FREQ pragma.

These functions can be used in the standard way when writing any program for BoostC.

void delay_us(unsigned char t)

(generated function) Delays execution for 't' micro seconds. Declared in boostc.h This function gets generated every time a project is linked and is controlled by the CLOCK_FREQ pragma. In some cases when clock frequency is too low it's not physically possible to generate this function. If that's the case linker will issue a warning.

void delay_10us(unsigned char t)

(generated function) Delays execution for 't*10' micro seconds. Declared in boostc.h This function gets generated every time a project is linked and is controlled by the CLOCK_FREQ pragma. In some cases, when clock frequency is too low, it's not physically possible to generate this function.
If that's the case, the linker will issue a warning.

void delay_100us(unsigned char t)

(generated function) Delays execution for 't*100' micro seconds. Declared in boostc.h This function gets generated every time a project is linked and is controlled by the CLOCK_FREQ pragma. In some cases, when clock frequency is too low, it's not physically possible to generate this function.
If that's the case, the linker will issue a warning.

void delay_ms(unsigned char t)

(generated function) Delays execution for 't' milli seconds. Declared in boostc.h This function gets generated every time a project is linked and is controlled by the CLOCK_FREQ pragma.

void delay_s(unsigned char t)

(generated function) Delays execution for 't' seconds. Declared in boostc.h This function gets generated every time a project is linked and is controlled by the CLOCK_FREQ pragma.

Note: The delays produced by these functions are generated using software delay loops, therefore the delay may actually be larger than that specified if the delay routine is interrupted by an interrupt.

System Libraries

A number of standard functions are included into BoostC installations. The number of such functions isn't static. It increases from release to release as new features are added. Most of these functions are declared in boostc.h (It's not recommended to include boostc.h directly into your code. Instead include system.h which in turn included boostc.h)

General purpose functions

clear_bit(var, num)

(macro) Clears bit 'num' in variable 'var'. Declared in boostc.h

set_bit(var, num)

(macro) Sets bit 'num' in variable 'var'. Declared in boostc.h

test_bit(var, num)

(macro) Tests if bit 'num' in variable 'var' is set. Declared in boostc.h

MAKESHORT(dst, lobyte, hibyte)

(macro) Makes a 16 bit long value (stored in 'dst') from two 8-bit long values (low byte 'lobyte' and high byte 'hibyte'). 'dst' must be a 16-bit long variable. Declared in boostc.h

```
  unsigned short res;
  MAKESHORT( res, adresl, adresh ); //make 16 bit value from adresh:adresl
registers and write it into variable 'res'
```

LOBYTE(dst, src)

(macro) Gets low byte from 'src' and writes it into 'dst'. Declared in boostc.h

HIBYTE(dst, src)

(macro) Gets high byte from 'src' and writes it into 'dst'. 'src' must be a 16-bit long variable. Declared in boostc.h

void nop(void)

(inline function) Generates one 'nop' instruction. Declared in boostc.h

void clear_wdt(void)

(inline function) Generates one 'clrwdt' instruction. Declared in boostc.h

void sleep(void)

(inline function) Generates one 'sleep' instruction. Declared in boostc.h

String and Character Functions

void strcpy(char *dst, const char *src)
void strcpy(char *dst, rom char *src)

void strncpy(char *dst, const char *src, unsigned char len)
void strncpy(char *dst, rom char *src, unsigned char len)

(function) Copies zero terminated string 'src' into destination buffer 'dst'. Destination buffer must be big enough for string to fit. Declared in string.h

unsigned char strlen(const char *src)
unsigned char strlen(rom char *src)

(function) Returns length of a string. Declared in string.h

signed char strcmp(const char *src1, const char *src2)
signed char strcmp(rom char *src1, const char *src2)
signed char strcmp(const char *src1, rom char *src2)
signed char strcmp(rom char *src1, rom char *src2)

signed char stricmp(const char *src1, const char *src2)
signed char stricmp(rom char *src1, const char *src2)
signed char stricmp(const char *src1, rom char *src2)
signed char stricmp(rom char *src1, rom char *src2)

(function) Compares two strings. Returns -1 if string #1 is less than string #2, 1 if string #1 is greater than string #2 or 0 is string #1 is same as string #2. Declared in string.h

signed char strncmp(char *src1, char *src2, unsigned char len)
signed char strncmp(rom char *src1, char *src2, unsigned char len)
signed char strncmp(char *src1, rom char *src2, unsigned char len)
signed char strncmp(rom char *src1, rom char *src2, unsigned char len)

signed char strnicmp(char *src1, char *src2, unsigned char len)
signed char strnicmp(rom char *src1, char *src2, unsigned char len)
signed char strnicmp(char *src1, rom char *src2, unsigned char len)
signed char strnicmp(rom char *src1, rom char *src2, unsigned char len)

(function) Compares first 'len' characters of two strings. Returns -1 if string #1 is less than string #2, 1 if string #1 is greater than string #2 or 0 is string #1 is same as string #2. Declared in string.h

void strcat(char *dst, const char *src)
void strcat(char *dst, rom char *src)

void strncat(char *dst, const char *src, unsigned char len)
void strncat(char *dst, rom char *src, unsigned char len)

(function) Appends zero terminated string 'src' to destination string 'dst'. Destination buffer must be big enough for string to fit. Declared in string.h

char* strpbrk(const char *ptr1, const char *ptr2)
char* strpbrk(const char *src, rom char *src)

unsigned char strcspn(const char *src1, const char *src2)
unsigned char strcspn(rom char *src1, const char *src2)
unsigned char strcspn(const char *src1, rom char *src2)
unsigned char strcspn(rom char *src1, rom char *src2)

(function) Locates the first occurrence of a character in the string that doesn't match any character in the search string. Declared in string.h

unsigned char strspn(const char *src1, const char *src2)
unsigned char strspn(rom char *src1, const char *src2)
unsigned char strspn(const char *src1, rom char *src2)
unsigned char strspn(rom char *src1, rom char *src2)

(function) Locates the first occurrence of a character in the string. Declared in string.h

char* strtok(const char *ptr1, const char *ptr2)
char* strtok(const char *src, rom char *src)

(function) Breaks string pointed into a sequence of tokens, each of which is delimited by a character from delimiter string. Declared in string.h

char* strchr(const char *src, char ch)

(function) Locates the first occurrence of a character in the string. Declared in string.h

char* strrchr(const char *src, char ch)

(function) Locates the last occurrence of a character in the string. Declared in string.h

char* strstr(const char *ptr1, const char *ptr2)
char* strstr(const char *src, rom char *src)

(function) Locates the first occurrence of a sub-string in the string. Declared in string.h

Conversion Functions

unsigned char sprintf(char* buffer, const char *format, unsigned int val)

Outputs a numerical value to a string in the specified format. The buffer must be long enough to hold the result. Only one numerical value can be output at a time. Declared in stdio.h.

Format specified in the format string with the following format:

%[flags][width][radix specifier]

Radix	Example output	Description
"%d"	"-120"	decimal signed integer
"%u"	"150"	decimal unsigned integer
"%o"	"773"	octal unsigned integer
"%X"	"ABF1"	hex unsigned integer
"%b"	"101101"	binary unsigned integer

Justification	Example output	Description
"%8d"	"231 "	left justified, padded to 8 characters length
"%016u"	"0000000000045102"	left justified, padded with zeroes to 16 characters length
"%-8b"	" 10"	right justified, padded 8 characters length

Display of sign	Example output	Description
"%+8d"	"+972 "	left justified, padded 8 characters length, signed always displayed
"% 8d"	" 765 "	left justified, padded 8 characters length, positive signed displayed as ' '
Display of sign only applies to signed decimal radix. Radix and field width added just to show complete format specification		

Implementation of field width is non standard - If a justification width is specified the width will be padded **or truncated** to match the width provided. The most significant digits and sign maybe truncated. Standard implementations do not truncate the output, which can cause unexpected buffer overrun.

int strtoi(const char buffer, char** endPtr, unsigned char radix)*

(Function) String to integer. A function that converts the numerical character string supplied into a signed integer (16 bit) value using the radix specified. Radix valid range 2 to 26.

buffer: Pointer to a numerical string.

endPtr: Address of a pointer. This is filled by the function with the address where string scan has ended. Allows determination of where there is the first non-numerical character in the string. Passing a NULL is valid and causes the end scan address not to be saved.

radix: The radix (number base) to use for the conversion, typical values: 2 (binary), 8 (octal), 10 (decimal), 16 (hexadecimal).

Return: The converted value.

long strtol(const char buffer, char** endPtr, unsigned char radix);*

(Function) String to long integer. A function that converts the numerical character string supplied into a signed long integer (32 bit) value using the radix specified. Radix valid range 2 to 26.

buffer: Pointer to a numerical string

endPtr: Address of a pointer. This is filled by the function with the address where string scan has ended. Allows determination of where there is the first non-numerical character in the string. Passing a NULL is valid and causes the end scan address not to be saved.

radix: The radix (number base) to use for the conversion, typical values: 2 (binary), 8 (octal), 10 (decimal), 16 (hexadecimal).

Return: The converted value.

int atoi(const char buffer)*

(Macro) ASCII to integer. A macro that converts the numerical character string supplied into a signed integer (16 bit) value using a radix of 10.

buffer: Pointer to a numerical string.

Return: The converted value.

Note: Macro implemented as: #define atoi(buffer) strtoi(buffer, NULL, 10)

long atol(const char buffer)*

(Macro) ASCII to long integer. A macro that converts the numerical character string supplied into a signed long integer (32 bit) value using a radix of 10.

buffer: Pointer to a numerical string.

Return: The converted value.

Note: Macro implemented as: #define atoi(buffer) strtoi(buffer, NULL, 10)

char itoa(int val, char* buffer, unsigned char radix)*

(Function) Integer to ASCII. function that converts an integer (16 bit) value into a character string.

char ltoa(long val, char* buffer, unsigned char radix)*

(Function) Long integer to ASCII. function that converts an long integer (32 bit) value into a character string.

Lightweight Conversion Functions

The standard conversion functions offer a lot of flexibility at the cost of ROM, RAM and execution time. For application that are short of RAM and ROM, or require shorter execution time, it maybe desirable to use the following lightweight functions.

void uitoa_hex(char buffer, unsigned int val, unsigned char digits)*

(Function) Unsigned integer to ASCII, hexadecimal representation. This function converts a 16 bit unsigned integer into a hex value with leading zeros. The number of digits is specified using by the *digits* parameter.

void uitoa_bin(char buffer, unsigned int val, unsigned char digits)*

(Function) Unsigned integer to ASCII, binary representation. This function converts a 16 bit unsigned integer into a binary value with leading zeros. The number of digits is specified using by the *digits* parameter.

void uitoa_dec(char buffer, unsigned int val, unsigned char digits)*

(Function) Unsigned integer to ASCII, decimal representation. This function converts and 16 bit unsigned integer into a decimal value with leading zeros. The number of digits is specified using by the *digits* parameter.

unsigned int atoui_hex(const char buffer)*

(Function) ASCII to unsigned integer, hexadecimal representation. This function converts a hexadecimal string value into 16 bit unsigned integer.

unsigned int atoui_bin(const char buffer)*

(Function) ASCII to unsigned integer, binary representation. This function converts a binary string value into 16 bit unsigned integer.

unsigned int atoui_dec(const char buffer)*

(Function) ASCII to unsigned integer, decimal representation. This function converts a decimal string value into 16 bit unsigned integer.

Character

char toupper(char ch)

(function) Converts lowercase character to uppercase. Declared in ctype.h

char tolower(char ch)

(function) Converts uppercase character to lowercase. Declared in ctype.h

char isdigit(char ch)

(function) Checks if character 'ch' is a digit. Returns non zero if this is a digit. Declared in ctype.h

char isalpha(char ch)

(function) Checks if character 'ch' is a letter. Returns non zero if this is a letter. Declared in ctype.h

char isalnum(char ch)

(function) Checks if character 'ch' is a letter or a digit. Returns non zero if this is a letter or a digit. Declared in ctype.h

char isblank(char ch)

(function) Returns a 1 if the argument is a standard blank character. All other inputs will return a 0. The following are the standard blank characters:
' ' (space) or '\t' (horizontal tab). Declared in ctype.h

char iscntrl(char ch)

(function) Returns a 1 if the argument is a valid control character. All other inputs will return a 0. Declared in ctype.h

char isgraph(char ch)

(function) Returns a 1 if the argument is a valid displayable ASCII character. All other inputs will return a 0. Declared in ctype.h

char islower(char ch)

(function) Returns a 1 if the argument is a valid lower-case ASCII letter. All other inputs will return a 0. Declared in ctype.h

char isprint(char ch)

(function) Returns a 1 if the argument is a valid printable ASCII character. All other inputs will return a 0. Declared in ctype.h

char ispunct(char ch)

(function) Returns a 1 if the argument is a valid punctuation character. All other inputs will return a 0. The following are the implemented punctuation characters:
! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~

char isspace(char ch)

(function) Returns a 1 if the argument is a standard white-space character. All other inputs will return a 0. Declared in ctype.h. The following are the standard white-space characters:

Character Description	Character ASCII code	Character Escape sequence
space	0x20	' '
horizontal tab	0x09	'\t'
vertical tab	0x0B	'\v'
newline	0x0A	'\n'

Character Description	Character ASCII code	Character Escape sequence
carriage return	0x0D	'\r'
form feed	0x0C	'\f'

char isupper(char ch)

(function) Returns a 1 if the argument is a valid upper-case ASCII letter. All other inputs will return a 0. Declared in ctype.h

char isxdigit(char ch)

(function) Returns a 1 if the argument is a valid hexadecimal character. All other inputs will return a 0. Declared in ctype.h

void* memchr(const void *ptr, char ch, unsigned char len)

(function) Locates the first character in memory. Declared in memory.h

signed char memcmp(const void *ptr1, const void *ptr2, unsigned char len)

(function) Compares memory. Declared in memory.h

void* memcpy(void *dst, const void *src, unsigned char len)

(function) Copies memory. Declared in memory.h

void* memmove(void *dst, const void *src, unsigned char len)

(function) Moves memory. Declared in memory.h

void* memset(void *ptr, char ch, unsigned char len)

(function) sets memory. Declared in memory.h

Miscellaneous Functions

unsigned short rand(void)

(function) Generates pseudo random number. Declared in rand.h Defined in rand.lib

void srand(unsigned short seed)

(function) Sets seed for pseudo random number generator. Declared in rand.h Defined in rand.lib

max(a, b)

(Macro) Returns the value of the argument with the largest value.

min(a, b)

(Macro) Returns the value of the argument with the smallest value.

I2C functions

i2c_init, i2c_start, i2c_restart, i2c_stop, i2c_read, i2c_write
(for more information look into i2c_driver.h and i2c_test.c files)

RS232 functions

uart_init, kbhit, getc, getch, putc, putch
(for more information look into serial_driver.h and serial_test.c files)

LCD functions

lcd_setup, lprintf, lcd_clear, lcd_write, lcd_funcmode, lcd_datamode
(for more information look into lcd_driver.h and lcd.c files)

Flash functions

short flash_read(short addr)

(function) Reads flash content from address 'addr'. Works with PIC16F87X devices. Declared in flash.h Defined in flash.pic16.lib

void flash_loadbuffer(short data)

(function) Stores 'data' in an internal buffer of 4 shorts long. Must be called four times to fill the internal buffer. Data in this buffer is used by flash_write to store data in flash. Works with PIC16F87X devices. Declared in flash.h Defined in flash.pic16.lib

void flash_write(short addr)

(function) Writes data from an internal buffer into flash at address 'addr'. The internal buffer that is 4 shorts long must be filled using 4 calls to flash_loadbuffer. Works with PIC16F87X devices. Declared in flash.h Defined in flash.pic16.lib

EEPROM functions

char eeprom_read(char addr)

(function) Reads eeprom content from address 'addr'. Works with PIC16F87X devices. Declared in eeprom.h Defined in eeprom.pic16.lib

void eeprom_write(char addr, char data)

(function) Writes 'data' into eeprom at address 'addr'. Works with PIC16F87X devices. Declared in eeprom.h Defined in eeprom.pic16.lib

ADC functions

short adc_measure(char ch)

(function) Reads ADC channel 'ch'. ADC must be initialized before using this function. Works with PIC16F devices that have ADC units. Declared in adc.h Defined in adc.pic16.lib

A sample ADC initialization can look like:

```
volatile bit adc_on @ ADCON0 . ADON; //AC activate flag

set_bit(adcon1, ADFM); // AD result needs to be right justified
set_bit(adcon1, PCFG0); // all analog inputs
set_bit(adcon1, PCFG1); // Vref+ = Vdd
set_bit(adcon1, PCFG2); // Vref- = Vss

set_bit(adcon0, ADCS1); // Select Tad = 32 * Tosc (this depends on the X-
tal here 10 MHz, should work up to 20 MHz)
clear_bit(adcon0, CHS0); // Channel 0
clear_bit(adcon0, CHS1); //
clear_bit(adcon0, CHS2); //
adc_on = 1; // Activate AD module
```

One wire bus functions

char oo_busreset()

(function) Resets the one wire bus. Declared in oo.h Defined in oo.pic16.lib and oo.pic18.lib

Here is a typical scenario how to use the one wire library:

```
// To be able to use the one wire library two global bit variables need to
be declared in the code.
// These are the variables that control port pin ised for one wire
communication. For example
// if the one wire interface is connected to pin 6 of port B the
declaration will look like this:

#define OO_PORT PORTB
#define OO_TRIS TRISB
#define OO_PIN 6

volatile bit oo_bus @ OO_PORT . OO_PIN;
volatile bit oo_bus_tris @ OO_TRIS . OO_PIN;

...

// Reset the one wire bus
oo_busreset();

// Start the conversion (non-blocking function)
oo_start_conversion();

// Wait for completion, you could do other stuff here
// But make sure that this function returns zero before
// reading the scratchpad
if( oo_wait_for_completion() )
{
  //handle conversion time out
}
```

```
// Read the scratchpad
if( oo_read_scratchpad() )
{
  //handle conversion error
}

// And extract the temperature information
short data = oo_get_data();
```

short oo_get_data()

(function) Reads data from one wire bus. Declared in oo.h Defined in oo.pic16.lib and oo.pic18.lib

char oo_read_scratchpad()

(function) Reads scratchpad. Declared in oo.h Defined in oo.pic16.lib and oo.pic18.lib

void oo_start_conversion()

(function) Starts conversion. Declared in oo.h Defined in oo.pic16.lib and oo.pic18.lib

char oo_conversion_busy()

(function) Checks if conversion is in progress. Returns 0 if no conversion is active. Declared in oo.h Defined in oo.pic16.lib and oo.pic18.lib

char oo_wait_for_completion()

(function) Waits for a conversion to complete. Returns 0 if conversion completed within 1 sec. Declared in oo.h Defined in oo.pic16.lib and oo.pic18.lib

Technical support

For technical support, 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://www.sourceboost.ipbhost.com/

BoostC Support Subscription

By buying a support subscription you will receive priority technical support via email. This ensures that your query or problem will be at the front of the queue and receive the highest priority attention.

BoostC Support Subscriptions are here:
http://www.sourceboost.com/Products/BoostC/BuyLicense/SupportSubscription.html

Licensing Issues

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

General Support

For general support issues, please mail support@sourceboost.com

Always Pleased To Hear From You!

We are always pleased to hear your comments, this helps us to satisfy your needs. Send mail to support@sourceboost.com or post your comments on the SourceBoost Forum.

Legal Information

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THE AUTHOR RESERVES THE RIGHT TO REJECT ANY LICENSE (REGISTRATION) REQUEST WITHOUT EXPLAINING THE REASONS WHY SUCH REQUEST HAS BEEN REJECTED. IN CASE YOUR LICENSE (REGISTRATION) REQUEST GETS REJECTED YOU MUST STOP USING THE SourceBoost IDE, BoostC, BoostC++, BoostBasic, C2C-plus, C2C++ and P2C-plus COMPILERS AND REMOVE THE WHOLE SourceBoost IDE INSTALLATION FROM YOUR COMPUTER.

Microchip, PIC, PICmicro and MPLAB are registered trademarks of Microchip Technology Inc.

BoostC and BoostLink are trademarks of SourceBoost Technologies.
Other trademarks and registered trademarks used in this document are the property of their respective owners.

http://www.sourceboost.com
Copyright© 2004-2006 Pavel Baranov

Copyright© 2004-2006 David Hobday