White Paper

An Le
John J. Oneill
Developer Products Division
Intel Corporation

Migrating Applications to Intel® Compilers for Mac OS*

Table of Contents

Introduction 4
Overview 4
Getting Started 4
Supported Features 4
Performance Features 4
Support for Apple Specific Features 4
Support for Apple-specific Linker Options 4
Building Applications 5
Invoking the Compiler from the Command line 5
Set the Environment Variables 5
Invoking the Compiler with icc or icpc 5
Command-line Syntax 5
Selecting Intel® C++ Compiler for Mac OS* from Xcode 2.2.1 6
Compatibility, Interoperability, and Standard Compliance 7
Building a Universal Binary with Intel® C++ Compiler for Mac OS* 7
Building Universal Binary From Xcode 2.2.1 7
Building Universal Binary from Command Line 8
Performance 8
Auto Vectorization at Default Optimization (-O2) 9
High Level Optimization (-O3) 9
Interprocedural Optimization (IPO) 9
Profile Guided Optimization 9
Full Support for OpenMP* 2.5 9
Auto Parallelization 9

Conclusion 10

References 10

Appendix A : Equivalent Intel® C++ Compiler Options for Metrowerks CodeWarrior* 9.0 For Mac OS* 11

Table 1: Preprocessing, Precompiling, and Input File Control Options 11

Table 2: C/C++ Language Options 12

Table 3: Code Optimization Options 13

Table 4: Debugging Control Options 13

Appendix B : Equivalent Intel® C++ Compiler options for IBM XL C/C++ Advanced Edition for Mac OS* X 14

Introduction

This document provides the background for developers who are migrating to the Intel® C++ Compiler 9.1 for Mac OS*. It discusses default behavior, and it highlights optimization and source changes that may be needed as part of the migration.

Overview

The Intel® C++ Compiler 9.1 for Mac OS* integrates with Apple's Xcode* 2.2.1 and is compatible with GNU C/C++ 4.0 to provide very high levels of optimization for Intel® processor technologies. In addition to performance and industry-compatibility gains, this version provides full support for processor-specific instruction sets available on Mac OS running on Intel® processors. Intel® software technologies such as code generation, optimization, and parallel processing, combined with compatibility features and support for language extensions, offer a robust feature set and excellent runtime performance.

In general, the source changes required to migrate to Intel C++ Compiler for Mac OS are the same as those for GNU C++ 4.0. In most cases, if you can build existing applications with GCC 4.0 in Xcode 2.2.1, you can rebuild the applications with the Intel C++ Compiler for Mac OS without source changes. Some applications may need minor coding changes, and build methods may need minor adjustments.

This document assumes that your project is imported to Xcode 2.2.1 or built from the command line. You may want to look at the following documents from the Apple developer site for more information:

- Porting CodeWarrior* Project to Xcode [1]
- GCC Porting Guide [2]

Getting Started

Supported Features

The Intel C++ Compiler for Mac OS runs on Intel processor-based Mac systems running Mac OS X* version 10.4.4. Key features supported by the Intel C++ Compiler for Mac OS include those discussed below.

Performance Features

- Auto-vectorization and support for compiler intrinsics for MMX, SSE, SSE2, and SSE3 instruction sets at default optimization (O2)
- Advanced floating-point model
- Interprocedural optimization (IPO)
- Profile-guided optimization (PGO)
- High-level language optimization (HLO)
- Support for OpenMP* 2.5
- Auto-parallelization
- Compiler-generated optimization reports

Support for Apple Specific Features

- Support for GNU inline ASM syntax
- Support for Microsoft MASM-style inline assembly format (-use-msasm)
- Support of Pascal strings (-fpascal-strings)
- Support for the weak_import attribute
- Support for Apple legacy alignment options (-malign-power, -malign-natural, -malign-mac68k)
- Support for Apple frameworks and Apple's C++ language extensions
- Interoperable with objective C, where GCC will compile object C source
- Universal binary support

Support for Apple-specific Linker Options

- Produce a Mach-O bundle format file with (-bundle)
- Produce Mach-O demand page executable format file (-execute)
- Specify file containing list of files to link (-filelist)
- Build output as flat namespace image; not default (-flat_namespace)
- Treat all dynamic libraries as flat namespace images (-force_flat_namespace)
- Build output as two level namespace image; default (-twolevel_namespace)

- Specify framework but mark all references to it as weak imports (-weak_framework)
- Build shared and static libraries (-dynamiclib)
- Generate non-pic code (-mdynamic-no-pic)

Building Applications

The Intel C++ Compiler for Mac OS integrates into the Xcode 2.2.1 IDE. If you are planning to build your application from the Xcode environment, you can use Xcode to import your projects. If you are building your application from the command line, you can invoke the Intel C++ Compiler for Mac OS directly. The following sections list the steps necessary to invoke the Intel® compiler from the command line or the Xcode environment.

Invoking the Compiler from the Command line

There are two necessary steps to invoke the Intel C++ Compiler for Mac OS from the command line:

1. Set the compiler environment.
2. Invoke the compiler.

Set the Environment Variables

Before you can operate the compiler, you must set the environment variables to specify locations for the various components. The Intel C++ Compiler for Mac OS installation includes shell scripts that you can use to set environment variables. With the default compiler installation, these scripts are:

<install-dir>/bin/iccvars.sh
or
<install-dir>/bin/iccvars.csh

To run an environment script, enter one of the following on the command line:

source <install-dir>/bin/iccvars.sh
or
source <install-dir>/bin/iccvars.csh

If you compile a program without 'sourcing' iccvars.sh, you will see the following error when you execute the compiled program:

./a.out: error while loading shared libraries:

libimf.so: cannot open shared object file: No such file or directory

Invoking the Compiler with icc or icpc

You can invoke the Intel C++ Compiler for Mac OS on the command line with either icc or icpc.

1. When you invoke the compiler with icc, the compiler builds C source files using C libraries and C include files. If you use icc with a C++ source file, it will be compiled as a C++ file. Use icc to link C object files.
2. When you invoke the compiler with icpc, the compiler builds C++ source files using C++ libraries and C++ include files. If you use icpc with a C source file, it will be compiled as a C++ file. Use icpc to link C++ object files.

Command-line Syntax

When you invoke the Intel C++ Compiler for Mac OS with icc or icpc, use the following syntax:

{icc|icpc} [options] file1 [file2 . . .]

Selecting Intel® C++ Compiler for Mac OS* from Xcode 2.2.1

GCC is the default Xcode* compiler for C, C++, and Objective-C. To use the Intel C++ Compiler for C or C++ in Xcode, do the following (see Figure 1):

1. Highlight the target you want to change in the Groups & Files list under the Target group.
2. Choose Get Info from the File menu or click the Info button in the toolbar.
3. Click Rules in the Target Info window.
4. Click the + button at the bottom left-hand corner of the Target Info window.
5. From the pull-down menu, choose C source files using the Intel® C/C++ Compiler.

Argument	Description
options	Indicates one or more command-line options. The compiler recognizes one or more letters preceded by a hyphen (-). This includes linker options.
file1, file2 . . .	Indicates one or more files to be processed by the compiler. You can specify more than one file. Use a space as a delimiter for multiple files.

Selecting Intel® C++ Compiler for Mac OS* from Xcode 2.2.1

GCC is the default Xcode* compiler for C, C++, and Objective-C. To use the Intel C++ Compiler for C or C++ in Xcode, do the following (see Figure 1):

1. Highlight the target you want to change in the Groups & Files list under the Target group.
2. Choose **Get Info** from the **File** menu or click the **Info** button in the toolbar.
3. Click **Rules** in the **Target Info** window.
4. Click the **+** button at the bottom left-hand corner of the **Target Info** window.
5. From the pull-down menu, choose **C source files using the Intel® C/C++ Compiler**.

Note: If you select a tool that does not support the source file type, then that source will be processed by a later rule that specifies that type. For example, even though Objective-C/C++ sources are derived from C sources, they will actually be built by GCC.

After you add a Target Info Rule, you can adjust the Intel compiler commands:

1. Select the **Target** from the **Groups & Files** list.
2. Choose **Get Info** from the **File** menu or click the **Info** button on the toolbar.
3. Select **Build**.

Note: Intel options default to the Release or Deployment configuration for both Release and Debug configurations.

Currently, the following General and GCC options are passed to the Intel compiler:

1. Header Search Paths – appended to the command line
2. Framework Paths – appended to the command line
3. GCC Preprocessor Macros – inserted before any user-supplied Defines

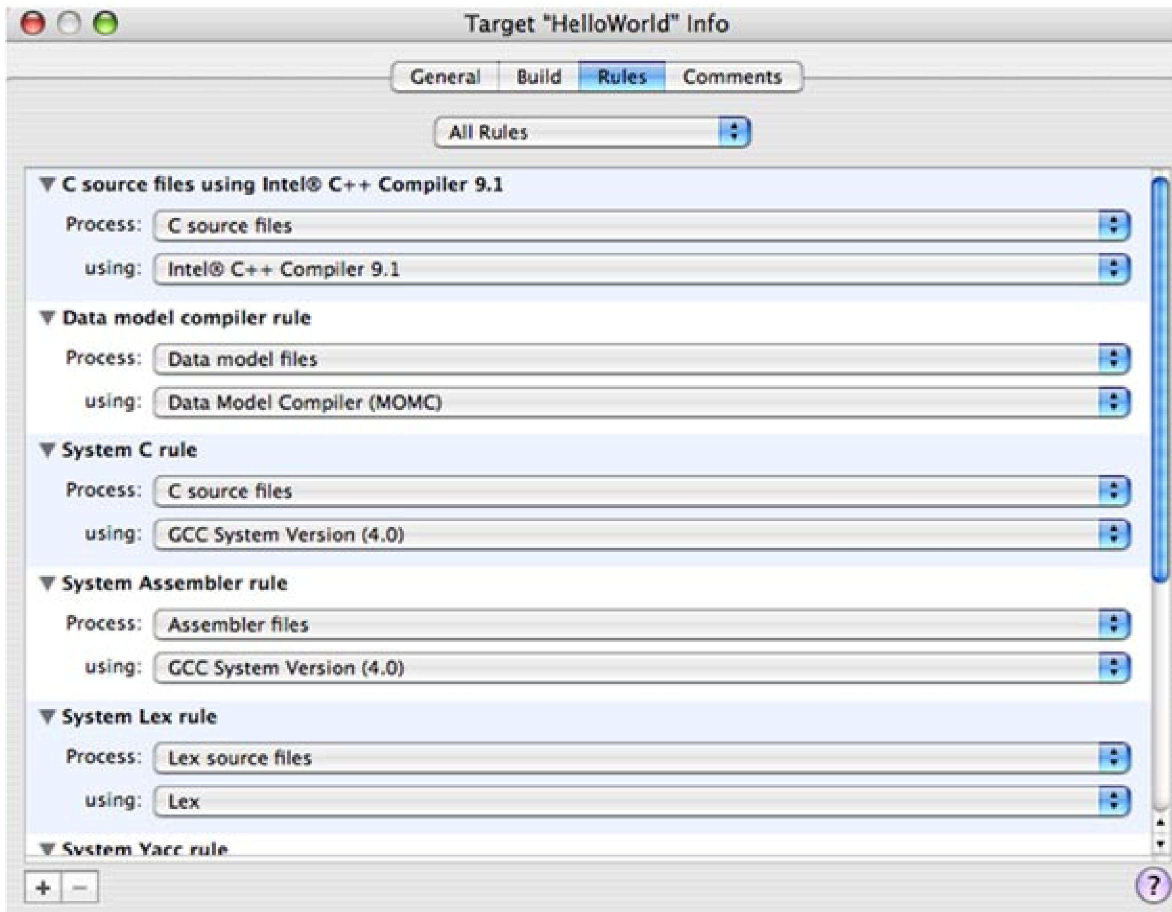


Figure 1. Compiler selection

Compatibility, Interoperability, and Standard Compliance

Mac OS X v10.4 introduced a new version of the GCC compiler – version 4.0. This new compiler provided significant improvements for the compilation of C and C++. With these improvements came stricter rules and better conformance to the C and C++ standards.

For flexibility, performance, and compatibility reasons, the Intel C++ Compiler for Mac OS is a good replacement for the GCC compilers. In addition to excellent source and binary compatibility with GNU C and C++ compilers version 4.0, the Intel C++ Compiler for Mac OS offers excellent performance on Intel processor-based Macs*.

When you compile existing code for the first time with the Intel C++ Compiler for Mac OS, you may see a lot of warnings in code that previously compiled cleanly. This is not an unusual occurrence. The Intel C++ Compiler for Mac OS is much stricter about code compliance than many of its peers.

Although the best way to remove warnings is to fix your code, this may seem like a daunting task for those who are just starting their transition. The Intel compilers support a full range of options to suppress warnings and soft errors. You can use these options to hide warnings until you can fix the errors reported by the compiler.

As mentioned earlier in this paper, Intel C++ Compiler for Mac OS supports the following Mac OS-specific features:

- Support for GNU inline ASM syntax
- Support for Microsoft MASM-style inline assembly format (-use-msasm)
- Support of Pascal strings (-fpascal-strings)
- Support weak_import attribute
- Support for Apple legacy alignment options (-malign-power, -malign-natural, -malign-mac68k)
- Support for Apple frameworks and Apple's C++ language extensions
- Interoperable with objective C where GCC will compile object C source
- Universal binary support

In addition to the above Mac OS-specific features, Intel C++ Compiler for Mac OS shares the GNU compiler source, binary, and command-line compatibility features described in the document, Intel® Compilers for Linux*: Compatibility with GNU Compilers, available at http://www.intel.com/software/products/compilers/clin.

Building a Universal Binary with Intel® C++ Compiler for Mac OS*

Building Universal Binary From Xcode 2.2.1

Produce a universal binary by configuring and building the program:

1. Open your project in Xcode 2.2.1 or later.
2. Select the **Intel C++ Compiler in Xcode**.
3. In the **Groups & Files** list, click the project name.
4. Click the **Info** button to open the **Info** window.
5. Select the **Architectures** setting and click **Edit**. In the sheet that appears, select the **PowerPC*** and **Intel** options, as shown in Figure 2.
6. Close the **Info** window.
7. Build and run the project.

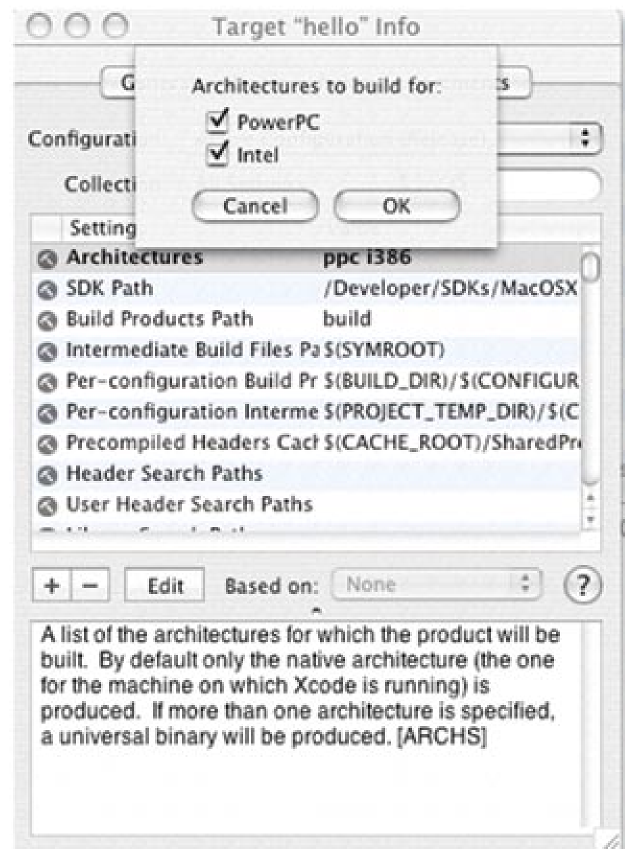


Figure 2. Option to Build Universal Binary

Building Universal Binary from Command Line

Produce a universal binary by configuring and building the program multiple times (very likely on multiple machines).

On both a Macintosh* based on Intel® and PowerPC* architectures, configure and build the program as follows:

1. Use Intel C++ Compiler for Mac OS to build binary for Intel based Macintosh.
2. Use GCC to build binary for PowerPC-based Macintosh.
3. Copy the resultant binaries to a single machine, with the names **hello-intel** and **hello-ppc**, respectively.
4. Use the lipo command to combine the two:
   lipo -create hello-intel hello-ppc -output hello

If your application does not behave as expected when you run it as a native binary on an Intel processor-based Macintosh computer, see "Troubleshooting"[3] in the Universal Binary Programming Guidelines[4].

If your application behaves as expected, do not assume that it also works on the other architecture. You need to test your application on both PowerPC Macintosh computers and Intel architecture-based Macintosh computers.

Performance

Key advantages of the Intel C++ Compiler for Mac OS are advanced optimization technology and performance-feature support. The Intel C++ Compiler for Mac OS offers aggressive optimization.

The next subsections describe a few of the key optimization phases available in the Intel C++ Compiler for Mac OS. Use of these optimizations can enable substantial performance gains in your application.

Auto Vectorization at Default Optimization (-O2)

Vectorization is an advanced optimization that analyzes loops and determines when it is safe and effective to execute several iterations of the loop in parallel by utilizing SIMD instructions such as SSE3. Vectorization automatically parallelizes code to maximize underlying processor capabilities. Features include support for advanced, dynamic data-alignment strategies, loop peeling to generate aligned loads, and loop unrolling to match the prefetch of a full cache line.

The values of the flags flush-to-zero (FTZ) and denormals-as-zero (DAZ) are set to **on** by default. Even at the cost of losing IEEE compliance, turning these flags on can significantly increase the performance of programs with denormal floating-point values on the most recent IA-32 processors.

Note these flags are set when **main** is compiled with the Intel C++ Compiler for Mac OS; otherwise, you would need to set these flags manually in main using intrinsics from **xmmintrin.h** and **pmmintrin.h** or use assembly code. For non-Intel compilers, you can set the flags manually with the following macros:

Feature	Examples
Enable FTZ	_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON)
Enable DAZ	_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON)

The prototypes for these macros are in **xmmintrin.h** (FTZ) and **pmmintrin.h** (DAZ).

This is the generally recommended optimization level. This option also enables the following:

- Inlining of intrinsics
- Some Intra-file interprocedural optimizations
- Constant propagation
- Copy propagation
- Dead-code elimination
- Global register allocation
- Global instruction scheduling and control speculation
- Loop unrolling
- Optimized code selection
- Partial redundancy elimination
- Strength reduction/induction variable simplification
- Variable renaming
- Exception handling optimizations
- Tail recursions
- Peephole optimizations
- Structure assignment lowering and optimizations
- Dead store elimination

High Level Optimization (-O3)

This optimization level enables O2 optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations. It enables optimizations for maximum speed, such as the following:

- Loop unrolling, including instruction scheduling
- Code replication to eliminate branches
- Padding the size of certain power-of-two arrays to allow more efficient cache use

The -O3 option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets.

Interprocedural Optimization (IPO)

This option enables interprocedural optimizations between files. When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

You cannot specify the names for the files that are created.

Compilers typically process one function at a time and in isolation from other functions in the program. A compiler with interprocedural optimization optimizes each function with detailed knowledge of other functions in the application, across multiple source files. Following are descriptions of several optimizations enabled by interprocedural optimization:

- Interprocedural constant propagation – constant values are propagated through function calls, particularly function call arguments.
- Arguments in registers – passing arguments in registers can reduce call/return overhead.
- Loop-invariant code motion – increased interprocedural information enables detection of code that can be safely moved outside of loop bodies.
- Dead code elimination – increased interprocedural information enables detection of code that may be proven unreachable.

Profile Guided Optimization

Profile-Guided Optimization (PGO) provides information to the compiler about areas of an application that are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application. For example, using PGO can often enable the compiler to make better decisions about function inlining, which increases the effectiveness of interprocedural optimizations.

Profile-guided optimization enables the compiler to learn from experience. Profile-guided optimization is a three-stage process:

1. Compilation of the application with instrumentation added
2. Profile generation phase where the application is executed and monitored.
3. A recompile, where the data collected during the first run aids optimization. A description of several code-size-influencing profile-guided optimizations follows:

- Basic block & function ordering – place frequently-executed blocks and functions together to take advantage of instruction cache locality.
- Aid inlining decisions – inline frequently-executed functions so the increase in code size is paid in areas of highest performance impact.
- Aid vectorization decisions – vectorize high trip count and frequently-executed loops so the increase in code size is mitigated by the increase in performance.

Full Support for OpenMP* 2.5

The Intel C++ Compiler for Mac OS supports the OpenMP* version 2.5 API specification and an automatic parallelization capability. OpenMP is the industry standard for portable multi-threaded application development, and it is effective at fine-grain (loop level) and large-grain (function level) threading. OpenMP directives are an easy and powerful way to convert serial applications into parallel applications, enabling potentially big performance gains from parallel execution on multi-core and symmetric multiprocessor systems.

The compiler performs transformations to generate multithreaded code based on the user's placement of OpenMP directives in the source program, making it easy to add threading to existing software. The Intel C++ Compiler for Mac OS supports all of the current industry-standard OpenMP directives, except WORKSHARE, and compiles parallel programs annotated with OpenMP directives.

Auto Parallelization

The auto-parallelization feature of the Intel C++ Compiler for Mac OS automatically translates serial portions of the input program into equivalent multithreaded code. The auto-parallelizer analyzes the dataflow of the loops in the application source code and generates multithreaded code for those loops that can safely and efficiently be executed in parallel.

This behavior enables the potential for high utilization of the parallel architecture found in symmetric multiprocessor (SMP) systems.

Automatic parallelization relieves the user from the following burdens:

- Dealing with the details of finding loops that are good work-sharing candidates
- Performing the dataflow analysis to verify correct parallel execution
- Partitioning the data for threaded code generation as is needed in programming with OpenMP directives

The parallel run-time support provides the same run-time features as found in OpenMP, such as handling the details of loop iteration modification, thread scheduling, and synchronization.

While OpenMP directives enable serial applications to transform into parallel applications quickly, a programmer must explicitly identify specific portions of the application code that contain parallelism and add the appropriate compiler directives.

Auto-parallelization, which is triggered by the -**parallel** option, automatically identifies those loop structures that contain parallelism. During compilation, the compiler automatically attempts to deconstruct the code sequences into separate threads for parallel processing. No other effort by the programmer is needed.

Conclusion

The Intel Compilers often enable substantial performance advantages for developers. The compilers also provide strict conformance to existing standards, thus helping to ensure that your code follows industry best practices. In addition, the Intel Compilers for Mac OS integrate with the Xcode 2.2.1 development environment and support many Apple-specific features.

References

- General information on Intel® software development tools, including the Intel C++ and Fortran compilers: http://www.intel.com/software/products
- Documentation, application notes, and source code for libraries, tools, and code examples: http://developer.intel.com/software/products/opensource/
- Information on Intel® Itanium® processor architecture: http://developer.intel.com/design/itanium/family
- Intel® Pentium® 4 processor information: http://developer.intel.com/design/pentium4/
- Introduction to Porting Codewarrior Project to Xcode: http://developer.apple.com/documentation/DeveloperTools/Conceptual/MovingProjectsToXcode/index.html
- Introduction to GCC porting guide: http://developer.apple.com/releasenotes/DeveloperTools/GCC40PortingReleaseNotes/index.html#//apple_ref/doc/uid/TP40002069
- Universal binary programming guidelines: http://developer.apple.com/documentation/MacOSX/Conceptual/universal_binary/

Appendix A : Equivalent Intel® C++ Compiler Options for Metrowerks CodeWarrior* 9.0 For Mac OS*

CodeWarrior*	Intel® Compiler	Comments
-c	**-c**	Global. Compile but do not link.
-D+ | -d[efine]name[=value]	**-D<name>[=<text>]**	Cased. Define symbol name to value if name=value is specified. Otherwise, define name to be 1.
-[no]defaults	**nostdinc**	Global. Passed to linker. Same as -[no]stdinc. Default.
-dis[assemble]	**-fcode-asm**	Global. Passed to all tools. Disassemble files and send result to stdout.
-E	**-E**	Global. Cased. Preprocess source files.
-EP	**-EP**	Global. Cased. Preprocess and strip out #line directives
-I+ | -i path	**-I<dir>**	Global. Cased. Append access path to current #include list. See -GCCincludes and -I-.
-M	**-M**	Global. Cased. Scan source files for dependencies and emit a Makefile. Do not generate object code.
-MM	**-MM**	Global. Cased. Like -M, but does not list system #include files.
-MD	**-MD**	Global. Cased. Like -M, but writes dependency map to a file and generates object code.
-MMD	**-MMD**	Global. Cased. Like -MD, but does not list system #include files.
-nolink	**-c**	Global. Compile only; do not link.
-o file | dir	**-o<file>**	Specify output file name or directory for object files, text files, and the linker's output file (if the linker is called after the compiler finishes).
-P	**-P, -F**	Global. Cased. Preprocess and send output to a file. Do not generate code.
-ppopt *keyword*[,...]		Specify options affecting the preprocessed output
The options for ***keyword*** are:		
-[no]line]	**-P, -F**	emit #line directives, else comments
-precompile file | dir	**-create-pch <file>**	Generate precompiled header from source. Write header to file, if specified. Place header in dir, if specified. If argument is "", write header to location specified in source code. If neither argument is defined, derive the header file name from the source file name. The driver can tell whether to precompile a file based on its extension. -precompile file source is therefore the same as -c -o file source.
-preprocess	**-E**	Global. Preprocess source files.
-S	**-S**	Global. Cased. Passed to all tools. Disassemble and send output to a file.
-[no]stdinc	**-nostdinc**	Global. Use standard system include paths (specified by the environment variable %MWCIncludes%. Add this option after all system -I paths. Default.
-U+ | -u[ndefine] name	**-U<name>**	Cased. Undefine the symbol name.

Table 1: Preprocessing, Precompiling, and Input File Control Options

CodeWarrior*	Intel® Compiler	Comments
-ansi *keyword*	**-ansi**	Specify ANSI conformance options, overriding the given settings.
The options for ***keyword*** are:		
Of	**(default)**	Same as -stdkeywords off, -enum min, and -strict off
on | relaxed	**-ansi**	Same as -stdkeywords on, -enum min, and -strict on
strict	**-strict-ansi**	Same as -stdkeywords on, -enum int, and -strict on
-char *keyword*		Set the default sign of the char data type.
The options for ***keyword*** are:		
Signed	**(default)**	char data items are signed. Default.
Unsigned	**-funsigned-char**	char data items are unsigned.
-dialect | -lang *keyword*		Specify source language.
The options for ***keyword*** are:		The options for keyword are:
c++	**-Kc++**	Treat source as C++ always.
-enum *keyword*		Specify the word size for enumerated types.
The options for ***keyword*** are:		
Min	**-fshort-enums**	Use minimum sized enums.
-inline *keyword*[,...]	**-Ob<n=0...2 >**	Specify inline options.
The options for ***keyword*** are:		
on | smart	**-Ob1**	Turn on inlining for __*inline keyword* functions. Default.
None | off	**-Ob0**	Turn off inlining.
Auto	**-Qip**	Auto-inline small functions (without inline explicitly specified).
All	**-Ob2**	Turn on aggressive inlining. This option is the same as -inline on, auto
-lang | -dialect | -lang *keyword*		Specify source language
The options for ***keyword*** are:		
C++	**-Kc++**	treat source as C++ always
C99	**-c99[-] (default on)**	compile with C99 extensions
-[no]multibyte[aware]	**-[no-]multibyte-chars**	Enable multi-byte character encoding for source text, comments, and strings. Use -enc
-RTTI on | off	**-frtti | -fno-rtti**	Select runtime typing information (for C++). Default is on.
-strict on | off	**-strict-ansi**	Specify ANSI strictness checking. Default is off.
-align *keyword*[,...]		Specify structure and array alignment.
The options for ***keyword*** are:		
1 | byte | packed	**-Zp1**	Byte alignment.
2	**-Zp2**	short alignment.
4	**-Zp4**	long alignment. Default.
8	**-Zp8**	Double word (floating double) alignment.
16	**-Zp16**	quadword alignment.
-profile on | off	**(See Profile Guided Optimization)**	Enable calls to an external profiling library. The default is off.

Table 2: C/C++ Language Options

CodeWarrior*	Intel® Compiler	Comments
-O	**-O2**	Same as -O2
-O+*keyword*[,...]		Cased. Control optimization. You can combine options. For example: -O4,p
The options for ***keyword*** are:		
0	**-O0**	Same as -opt off
1	**-O1**	Same as -opt level=1
2	**-O2**	Same as -opt level=2
3	**-O3**	Same as -opt level=3
P	**-Ot**	Same as -opt speed
S	**-Os**	Same as -opt space
-opt *keyword*[,...]		Specify code optimization options.
The options for ***keyword*** are:		
off | none	**-O0**	Suppress all optimizations. Default.
On	**-O2**	Same as -opt level=2
l[evel] = num		Set optimization level.
0	**-O0**	Global register allocation only for temporary values.
1	**-O1**	Adds dead code elimination, branch and arithmetic optimizations, expression simplification, and peephole optimization.
2	**-O2**	Adds common subexpression elimination, copy and expression propagation, stack frame compression, stack alignment, and fast floating-point to integer conversions.
3	**-O3**	Adds dead store elimination, live range splitting, loop-invariant code motion, strength reduction, loop transformations, loop unrolling (with -opt speed only), loop vectorization, lifetime-based register allocation, and instruction scheduling.

Table 3: Code Optimization Options

All optimization options besides –opt off | on | all | space | speed | level =... are for backwards compatibility. You can supersede other optimization options by using -opt level=xxx

CodeWarrior*	Intel® Compiler	Comments
-g	**-g**	Global. Cased. Generate symbolic debugging information. Same as -sym on

Table 4: Debugging Control Options

Appendix B : Equivalent Intel® C++ Compiler options for IBM XL C/C++ Advanced Edition for Mac OS* X

IBM XL C/C++ Option	Description	**Intel® C++ Compiler Option**	Comments
+ (plus sign)	Compiles any file, filename.nnn, as a C++ language file, where nnn is any suffix other than **.o**, **.a**, **or .s**.	-x c++	Not an exact match, -x c++ applies to all files after it is used on the command line, not just nnn extension files.
# (pound sign)	Traces the compilation without doing anything.	-#	
aggrcopy	Enables destructive copy operations for structures and unions.		
alias	Specifies which type-based aliasing is to be used during optimization.	-[no-]ansi-alias	
align	Specifies what aggregate alignment rules the compiler uses for file compilation.	-malign-power; -malign -natural; -malign-mac68k	
alloca	Substitutes inline code for calls to function **alloca** as if **#pragma alloca** directives are in the source code.		
altivec	Enables compiler support for AltiVec(TM) data types.		
ansialias	Specifies whether type-based aliasing is to be used during optimization.	-ansi-alias	
arch	Specifies the architecture on which the executable program will be run.		
assert	Instructs the compiler to apply aliasing assertions to your compilation unit.		
attr	Produces a compiler listing that includes an attribute listing for all identifiers.		
B	Determines substitute path names for the compiler, assembler, linkage editor, and preprocessor.	-B<dir>	
bitfields	Specifies if bitfields are signed.	-funsigned-bitfields	signed bitfields are default
bundle	Instructs the linker to create a bundle.	-bundle	direct to linker option

IBM XL C/C++ Option	Description	Intel® C++ Compiler Option	Comments
bundle_loader	Specifies the name of a bundle loader program.		
C	Preserves comments in preprocessed output.	-C	
c	Instructs the compiler to pass source files to the compiler only.	-c	
c_stdinc	Changes the standard search location for the C headers.		
cache	Specify a cache configuration for a specific execution machine.		
chars	Instructs the compiler to treat all variables of type **char** as either **signed** or **unsigned**.	-funsigned-char	signed chars is the default
check	Generates code which performs certain types of run-time checking.		
cinc	Include files from specified directories have the tokens **extern "C" {** inserted before the file, and **}** appended after the file.		
common	Controls where uninitialized global variables are allocated.		
compact	When used with optimization, reduces code size where possible, at the expense of execution speed.	-O1	
complexgccincl	Instructs the compiler to internally wrap **#pragma complexgcc(on)** and **#pragma complexgcc(pop)** directives around include files found in specified directories.		
cpluscmt	Use this option if you want C++ comments to be recognized in C source files.		default
cpp_stdinc	Changes the standard search location for the C++ headers.		
crt	Instructs the linker to use the standard system startup files at link time.	-nostartfiles	
D	Defines the identifier name as in a **#define** preprocessor directive.	-D	
mbcs, dbcs	Use the **-qdbcs** option if your program contains multibyte characters.	-[no-]multibyte-chars	
dbxextra	Specifies that all **typedef** declarations, **struct**, **union**, and **enum** type definitions are included for debugger processing.		

IBM XL C/C++ Option	Description	Intel® C++ Compiler Option	Comments
digraph	Enables the use of digraph character sequences in your program source.		
dollar	Allows the $ symbol to be used in the names of identifiers.		
E	Instructs the compiler to preprocess the source files.	-E	
eh	Controls exception handling.	-f[no-]exceptions	
enum	Specifies the amount of storage occupied by the enumerations.		
F	Names an alternative configuration file for the compiler.		
flag	Specifies the minimum severity level of diagnostic messages to be reported.	-w<int>	Controls the warning level
float	Specifies various floating point options to speed up or improve the accuracy of floating point operations.		
flttrap	Generates extra instructions to detect and trap floating point exceptions.		
framework	Names a framework to link to.	-framework	
frameworkdir	Adds a user-defined framework directory to the header file search path.	-F<dir>	
fullpath	Specifies what path information is stored for files when you use the **-g** option.		
g	Generates debugging information for use by a debugger.	-g	
gcc_c_stdinc	Changes the standard search location for the GCC headers.	GCCROOT environment variable	Can perform the same thing, but it is desired to point to a different GCC installation using -gcc-name
gcc_cpp_stdinc	Changes the standard search location for the g++ headers.	GXX_INCLUDE environment variable	
genproto	Produces ANSI prototypes from K&R function definitions.		
halt	Instructs the compiler to stop after the compilation phase when it encounters errors of specified severity or greater.		
haltonmsg	Instructs the compiler to stop after the compilation phase when it encounters a specific error message.		
hot	Instructs the compiler to perform high-order transformations on loops and array language during optimization, and to to pad array dimensions and data objects to avoid cache misses.	-O3	Our High Level optimizations, not exactly equivalent

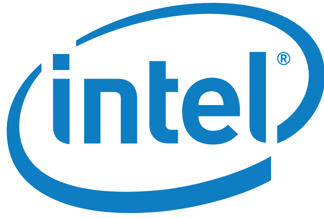
IBM XL C/C++ Option	Description	Intel® C++ Compiler Option	Comments
I	Specifies an additional search path for **#include** filenames that do not specify an absolute path.	-I	
idirfirst	Specifies the search order for files included with the **#include "** *file_ name*" directive.		
ignerrno	Allows the compiler to perform optimizations that assume **errno** is not modified by system calls.	-f[no-]math-errno	
ignprag	Instructs the compiler to ignore certain pragma statements.		
info	Produces informational messages.		
initauto	Initializes automatic storage to a specified two-digit hexadecimal byte value.		
inline	Attempts to inline functions instead of generating calls to a function.	-finline	
ipa	Turns on or customizes a class of optimizations known as interprocedural analysis (IPA).	-ipo	
isolated_call	Specifies functions in the source file that have no side effects.		
keyword	Controls whether a specified string is treated as a keyword or an identifier.		
L	Searches the specified directory at link time for library files specified by the -l option.	-L	
l	Searches a specified library for linking.	-l	
langlvl	Selects the C or C++ language level for compilation.		dependent on if icc or icpc is used or if a C++ file is used on the icc command line
lib	Instructs the compiler to use the standard system libraries at link time.		
libansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.		
linedebug	Generates abbreviated line number and source file name information for the debugger.		
list	Produces a compiler listing that includes an object listing.		
listopt	Produces a compiler listing that displays all options in effect.		
longlong	Allows **long long** types in your program.		

IBM XL C/C++ Option	Description	Intel® C++ Compiler Option	Comments
M	Creates an output file that contains targets suitable for inclusion in a description file for the **make** command.		
ma	Substitutes inline code for calls to function **alloca** as if **#pragma alloca** directives are in the source code.		
macpstr	Converts Pascal string literals into null-terminated strings where the first byte contains the length of the string.	-f[no-]pascal-strings	
makedep	Creates an output file that contains targets suitable for inclusion in a description file for the **make** command.		
maxerr	Instructs the compiler to halt compilation when a specified number of errors of specified or greater severity is reached.	-wn<n>	
maxmem	Limits the amount of memory used for local tables of specific, memory-intensive optimizations.		
mbcs, dbcs	Use the **-qmbcs** option if your program contains multibyte characters.	-[no-]multibyte-chars	
mkshrobj	Creates a shared object from generated object files.	-shared	
O, optimize	Optimizes code at a choice of levels during compilation.	-O1, -O2, -O3	
o	Specifies an output location for the object, assembler, or executable files created by the compiler.	-o	
P	Preprocesses the C or C++ source files named in the compiler invocation and creates an output preprocessed source file for each input source file.	-P	
p	Sets up the object files produced by the compiler for profiling.	-prof-gen	
path	Constructs alternate program and path names.		
pdf1, pdf2	Tunes optimizations through Profile-Directed Feedback.	-prof-use	
pg	Sets up the object files for profiling.	-pg	
phsinfo	Reports the time taken in each compilation phase.		
pic	Instructs the compiler to generate Position-Independent Code suitable for use in shared libraries.	-f[no-]pic	
print	**-qnoprint** suppresses listings.		

IBM XL C/C++ Option	Description	Intel® C++ Compiler Option	Comments
priority	Specifies the priority level for the initialization of static constructors		
proto	Assumes all functions are prototyped.		
Q	Attempts to inline functions instead of generating calls to a function.	-Ob2	
r	Produces a relocatable object.	-r	
report	Instructs the compiler to produce transformation reports that show how program loops are optimized.	-opt-report <level>	
ro	Specifies the storage type for string literals.		
roconst	Specifies the storage location for constant values.		
rtti	Generates run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.	-f[no-]rtti	
rwvftable	Instructs the compiler to place virtual function tables into read/write memory.		
s	Strips symbol table.		
showinc	Used with **-qsource** to selectively show user header files (includes using " ") or system header files (includes using < >) in the program source listing.	-MD	Not an exact match, -MD compiles and creates a .d file containing the include files
smallstack	Instructs the compiler to reduce the size of the stack frame.		
source	Produces a compiler listing and includes source code.		
sourcetype	Instructs the compiler to treat all source files as if they are the source type specified by this option, regardless of actual source filename suffix.	-x c++; -x c; -x c-header; -x cpp-output; -x c++-cpp-output; -x assembler; -x assembler-with-cpp; -x none	
spill	Specifies the size of the register allocation spill area.		
srcmsg	Adds the corresponding source code lines to the diagnostic messages in the **stderr** file.		
staticinline	Controls whether inline functions are treated as static or extern.		
statsym	Adds user-defined, non-external names that have a persistent storage class to the name list.		

IBM XL C/C++ Option	Description	Intel® C++ Compiler Option	Comments
stdframework	Determines if system default framework directories are searched for header files.		
stdinc	Specifies which files are included with **#include** <*file_name*> and **#include** *"file_name"* directives.		
strict	Turns off aggressive optimizations of the **-O3** option that have the potential to alter the semantics of your program.		
strict_induction	Disables loop induction variable optimizations that have the potential to alter the semantics of your program.		
suppress	Specifies compiler message numbers to be suppressed.		
symtab	Determines what information appears in the symbol table.		
syntaxonly	Causes the compiler to perform syntax checking without generating an object file.	-syntax	
t	Adds the prefix specified by the **-B** option to designated programs.		
tabsize	Changes the length of tabs as perceived by the compiler.		
tempinc	Generates separate include files for template functions and class declarations, and places these files in a directory which can be optionally specified.		
templaterecompile	Helps manage dependencies between compilation units that have been compiled using the **-qtemplateregistry** compiler option.		
templateregistry	Maintains records of all templates as they are encountered in the source and ensures that only one instantiation of each template is made.		
tempmax	Specifies the maximum number of template include files to be generated by the **tempinc** option for each header file.		
threaded	Indicates that the program will run in a multi-threaded environment.		default
tmplparse	Controls whether parsing and semantic checking are applied to template definition implementations.		
trigraph	Enables the use of trigraph character sequences in your program source.		
tune	Specifies the architecture for which the executable program is optimized.	-tune	

IBM XL C/C++ Option	Description	Intel® C++ Compiler Option	Comments
U	Undefines a specified identifier defined by the compiler or by the **-D** option.	-U	
unroll	Unrolls inner loops in the program.	-funroll	
unwind	Informs the compiler that the application does not rely on any program stack unwinding mechanism.		
upconv	Preserves the **unsigned** specification when performing integral promotions.		
V	Instructs the compiler to report information on the progress of the compilation in a command-like format.		
v	Instructs the compiler to report information on the progress of the compilation.	-V	
vftable	Controls the generation of virtual function tables.		
vrsave	Controls function prolog and epilog code necessary to maintain the VRSAVE register.		
W	Passes the listed words to a designated compiler program.		
w	Requests that warning messages be suppressed.	-w	
warnfourcharconsts	Enable warning of four-character constants in program source.		
xcall	Generates code to static routines within a compilation unit as if they were external calls.		
xref	Produces a compiler listing that includes a cross-reference listing of all identifiers.		
y	Specifies the compile-time rounding mode of constant floating-point expressions.		
Z	Specifies a search path for library names.		



For product and purchase information visit:
www.intel.com/software/products

Intel, the Intel logo, Intel. Leap ahead. and Intel. Leap ahead. logo, Pentium, Intel Core, and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Copyright © 2006, Intel Corporation. All Rights Reserved.

0506/DAM/ITF/PP/500 312568-001

---

[1] http://developer.apple.com/documentation/DeveloperTools/Conceptual/MovingProjectsToXcode/index.html

[2] http://developer.apple.com/releasenotes/DeveloperTools/GCC40PortingReleaseNotes/index.html#//apple_ref/doc/uid/TP40002069

[3] http://developer.apple.com/documentation/MacOSX/Conceptual/universal_binary/universal_binary_compiling/chapter_2_section_4.html#//apple_ref/doc/uid/TP40002217-CH206-CJBCIFEC

[4] http://developer.apple.com/documentation/MacOSX/Conceptual/universal_binary/universal_binary_compiling/chapter_2_section_4.html