

IBM XL C/C++ for Multicore Acceleration for Linux,
V10.1



Compiler Reference

Version 10.1

IBM XL C/C++ for Multicore Acceleration for Linux,
V10.1



Compiler Reference

Version 10.1

Note

Before using this information and the product it supports, read the information in “Notices” on page 371.

First edition

This edition applies to IBM XL C/C++ for Multicore Acceleration for Linux on Power Systems, V10.1 and IBM XL C/C++ for Multicore Acceleration for Linux on x86 Systems, V10.1 (Programs 5724-T42 & 5724-T43), and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© **Copyright International Business Machines Corporation 1996, 2008.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this information vii

Who should read this information.	vii
How to use this information	vii
How this information is organized	viii
Conventions.	viii
Related information.	xi
IBM XL C/C++ information.	xii
Standards and specifications	xiii
Other IBM information	xiii
Other information	xiii
Technical support	xiii
How to send your comments	xiv

Chapter 1. Compiling and linking applications 1

Invoking the compiler	1
Command-line syntax	3
Types of input files	4
Types of output files.	5
Specifying compiler options	6
Specifying compiler options on the command line	7
Specifying compiler options in a configuration file	8
Specifying compiler options in program source files	8
Resolving conflicting compiler options.	9
Reusing GNU C/C++ compiler options with ppugxlc , ppugxlc++ , spugxlc , and spugxlc++	10
ppugxlc , ppugxlc++ , spugxlc , or spugxlc++ syntax	11
Preprocessing.	12
Directory search sequence for include files	12
Linking.	13
Order of linking	14
Redistributable libraries	14
Compiler messages and listings.	15
Compiler messages	15
Compiler return codes.	17
Compiler listings	18

Chapter 2. Configuring compiler defaults 21

Setting environment variables	21
Compile-time and link-time environment variables	22
Runtime environment variables.	22
Using custom compiler configuration files	24
Creating custom configuration files	25
Configuring the ppugxlc , ppugxlc++ , spugxlc , or spugxlc++ option mapping	27

Chapter 3. Compiler options reference 31

Summary of compiler options by functional category	31
Output control	31

Input control	32
Language element control	33
Template control (C++ only).	34
Floating-point and integer control	35
Object code control	35
Error checking and debugging	37
Listings, messages, and compiler information	39
Optimization and tuning	41
Linking.	44
Portability and migration.	45
Compiler customization	45
Individual option descriptions	46
+ (plus sign) (C++ only)	47
# (pound sign)	48
-q32, -q64 (PPU only)	49
-qabi_version (C++ only)	49
-qaggrcopy	50
-qalias	51
-qalign	53
-qalloca, -ma (C only)	55
-qaltivec	56
-qarch	57
-qasm	58
-qasm_as (PPU only)	60
-qatomic_updates (SPU only)	61
-qattr	62
-B	63
-qbitfields	64
-c.	65
-C, -C!	66
-qcache (PPU only)	67
-qchars	69
-qcheck.	70
-qcinc (C++ only)	71
-qcommon (PPU only).	72
-qcompact	74
-qcomplexgccincl	74
-qcpluscmt (C only)	76
-qcrt.	77
-qc_stdinc (C only)	78
-qcpp_stdinc (C++ only)	79
-D	80
-qdataimported, -qdatalocal, -qtocdata (PPU only)	81
-qdbxextra (C only).	82
-qdigraph	83
-qdirectstorage (PPU only)	84
-qdollar.	85
-qdump_class_hierarchy (C++ only)	85
-e.	86
-E	87
-qea32, -qea64 (SPU only).	88
-qeh (C++ only) (PPU only)	89
-qenum.	90
-qenablevmx (PPU only)	94
-F.	94
-qfdpr	96

-qflag	97	-qprocimported, -qprocllocal, -qprocunknown (PPU only)	195
-qfloat	98	-qproto (C only)	197
-qfltrap (PPU only)	103	-Q, -qinline	198
-qformat	105	-r	200
-qfullpath	106	-R	201
-qfuncsect	107	-qreport	202
-g	108	-qreserved_reg	203
-qgcc_c_stdinc (C only)	109	-qro (PPU only)	204
-qgcc_cpp_stdinc (C++ only)	110	-qroconst (PPU only)	206
-qgenproto (C only)	111	-qrtti (C++ only) (PPU only)	207
-qhalt	113	-s	208
-qhaltonmsg (C++ only)	114	-S	208
-qhot	115	-qsaveopt.	209
-I	118	-qshowinc	211
-qidirfirst	119	-qshowmacros	212
-qignerno	120	-qsmallstack	213
-qignprag.	121	-qsmmp	214
-qinclude	122	-qsource	218
-qinfo	124	-qsourcecetype.	219
-qinitauto.	130	-qspill	220
-qinlgue (PPU only)	131	-qsrcmsg (C only)	221
-qinline	132	-qstaticinline (C++ only).	222
-qipa	132	-qstaticlink	223
-qisolated_call	141	-qstatsym.	224
-qkeepinlines (C++ only)	143	-qstdinc	225
-qkeepparm	144	-qstdmain (SPU only).	226
-qkeyword	145	-qstrict	227
-l	146	-qstrict_induction	231
-L	147	-qsuppress	231
-qlanglvl	148	-qswcache_size (SPU only)	233
-qlargepage	160	-qsyntab (C only)	234
-qlib	161	-qsyntonly (C only)	235
-qlibansi	162	-t	236
-qlinedebug	163	-qtabsize	237
-qlist	164	-qtbtable (PPU only)	238
-qlistopt	165	-qtempinc (C++ only).	239
-qlonglit (PPU only)	166	-qtemplatedepth (C++ only)	240
-qlonglong	167	-qtemplaterecompile (C++ only)	241
-ma (C only).	168	-qtemplateregistry (C++ only)	242
-qmakedep, -M.	168	-qtempmax (C++ only)	243
-qmaxerr	170	-qthreaded (PPU only)	244
-qmaxmem	172	-qtimesteps	245
-qmbcs, -qdbcs	173	-qtls (PPU only)	245
-MF	174	-qtmplinst (C++ only)	247
-qminimaltoc (PPU only)	175	-qtmplparse (C++ only)	248
-qmkshrobj (PPU only)	176	-qtocdata (PPU only)	249
-qnewcheck (C++ only)	177	-qtrigraph	249
-o	178	-qtune	250
-O, -qoptimize	179	-U	251
-qoptdebug	182	-qunroll	252
-p, -pg, -qprofile (PPU only)	183	-qunwind.	255
-P	184	-qupconv (C only).	255
-qpath	185	-qutf	256
-qpdf1, -qpdf2 (PPU only)	187	-v, -V	257
-qphsinfo	189	-qversion	258
-qpics	190	-qvrsave (PPU only)	259
-qppline	191	-w	261
-qprefetch (PPU only)	192	-W	262
-qprint	193	-qwarn64 (PPU only).	263
-qpriority (C++ only)	194	-qxcall.	264

-qxref	265
-y	266
-Z	268
Chapter 4. Compiler pragmas	
reference	269
Pragma directive syntax	269
Scope of pragma directives	270
Summary of compiler pragmas by functional category	270
Language element control	271
C++ template pragmas	271
Floating-point and integer control	271
Error checking and debugging.	271
Listings, messages and compiler information	272
Optimization and tuning	272
Object code control	273
Portability and migration	273
Compiler customization	274
Individual pragma descriptions	274
#pragma align	274
#pragma alloca (C only).	274
#pragma block_loop	274
#pragma chars	278
#pragma comment.	278
#pragma complexgcc	279
#pragma define, #pragma instantiate (C++ only)	279
#pragma disjoint	280
#pragma do_not_instantiate (C++ only).	281
#pragma enum	282
#pragma execution_frequency	282
#pragma expected_value	284
#pragma hashome (C++ only).	285
#pragma ibm snapshot	286
#pragma implementation (C++ only)	287
#pragma info	288
#pragma ishome (C++ only)	288
#pragma isolated_call	288
#pragma langlvl (C only)	288
#pragma leaves.	288
#pragma loopid	289
#pragma map	290
#pragma mc_func	293
#pragma nosimd	294
#pragma novector	294
#pragma options	294
#pragma option_override	296
#pragma pack	298
#pragma priority (C++ only)	302
#pragma reachable	302
#pragma reg_killed_by	303
#pragma report (C++ only).	304
#pragma STDC cx_limited_range	305
#pragma stream_unroll	306
#pragma strings	307
#pragma unroll.	307
#pragma unrollandfuse	308
#pragma weak	309
Pragma directives for parallel processing	312

Chapter 5. Compiler predefined macros	325
General macros.	325
Macros indicating the XL C/C++ compiler product	326
Macros related to the platform	327
Macros related to compiler features	328
Macros related to compiler option settings.	328
Macros related to architecture settings	330
Macros related to language levels	331

Chapter 6. Macros independent of compiler options	337
--	------------

Chapter 7. Compiler built-in functions	339
Fixed-point built-in functions	339
Absolute value functions	340
Assert functions	340
Count zero functions	340
Load functions	341
Multiply functions.	341
Population count functions.	342
Rotate functions	342
Store functions	344
Trap functions	344
Floating-point built-in functions	345
Absolute value functions	345
Conversion functions.	346
FPSCR functions	347
Multiply-add/subtract functions	349
Reciprocal estimate functions	350
Select functions.	350
Square root functions.	351
Software division functions.	351
Store functions	352
Synchronization and atomic built-in functions (PPU only)	352
Check lock functions	352
Clear lock functions	354
Compare and swap functions	355
Fetch functions	356
Load functions	357
Store functions	358
Synchronization functions	359
Cache-related built-in functions (PPU only)	360
Data cache functions	360
Prefetch functions	361
Block-related built-in functions	361
__bcopy	361
Miscellaneous built-in functions	362
Optimization-related functions	362
Move to/from register functions	363
Memory-related functions	365
Built-in functions for parallel processing	366
OpenMP built-in functions	366

Notices	371
Trademarks and service marks	373

Index	375
------------------------	------------

About this information

This information is a reference for the IBM® XL C/C++ for Multicore Acceleration for Linux®, V10.1 compiler. Although it provides information on compiling and linking applications written in C and C++, it is primarily intended as a reference for compiler command-line options, pragma directives, predefined macros, built-in functions, environment variables, and error messages and return codes.

Who should read this information

This information is for experienced C or C++ developers who have some familiarity with the XL C/C++ compilers or other command-line compilers on UNIX® operating systems. It assumes thorough knowledge of the C or C++ programming language, and basic knowledge of operating system commands. Although this information is intended as a reference guide, programmers new to XL C/C++ can still find information in it on the capabilities and features unique to the XL C/C++ compiler.

How to use this information

Unless indicated otherwise, all of the text in this reference pertains to both C and C++ languages. Where there are differences between languages, these are indicated through qualifying text and icons, as described in “Conventions” on page viii. Additionally, unless indicated otherwise, all of the text in this information pertains to compilation targeting both the Power Processor Unit (PPU) and Synergistic Processor Unit (SPU).

XL C/C++ provides separate invocation commands depending on whether you are compiling for the PPU or SPU. PPU-targeted compilation uses commands prefixed with **ppu**; SPU-targeted compilation uses commands prefixed with **spu**. In addition to these prefixes, other forms of the invocation commands are provided that use different default compiler settings. These are described in “Invoking the compiler” on page 1. However, for convenience, this information may use the invocation commands **xl**c and **xl**c++ to describe the actions of the C and C++ compiler, respectively. Similarly, to refer to the GCC-option utilities, **ppug**xl**c**, **spug**xl**c**, **ppug**xl**c++** and **spug**xl**c++**, this information may use the simplified forms **g**xl**c** and **g**xl**c++** only.

While this information covers topics on configuring the compiler environment, and compiling and linking C or C++ applications using XL C/C++ compiler, it does not include the following topics:

- Compiler installation: see the *XL C/C++ Installation Guide* for information on installing XL C/C++.
- The C or C++ programming languages: see the *XL C/C++ Language Reference* for information on the syntax, semantics, and IBM implementation of the C or C++ programming languages.
- Programming topics: see the *XL C/C++ Optimization and Programming Guide* for detailed information on developing applications with XL C/C++, with a focus on program portability and optimization.

How this information is organized

Chapter 1, “Compiling and linking applications,” on page 1 discusses topics related to compilation tasks, including invoking the compiler, preprocessor, and linker; types of input and output files; different methods for setting include file path names and directory search sequences; different methods for specifying compiler options and resolving conflicting compiler options; how to reuse GNU C/C++ compiler options through the use of the compiler utilities **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++**; and compiler listings and messages.

Chapter 2, “Configuring compiler defaults,” on page 21 discusses topics related to setting up default compilation settings, including setting environment variables, customizing the configuration file, and customizing the **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++** option mappings.

Chapter 3, “Compiler options reference,” on page 31 begins with a summary of options according to functional category, which allows you to look up and link to options by function; and includes individual descriptions of each compiler option sorted alphabetically.

Chapter 4, “Compiler pragmas reference,” on page 269 begins with a summary of pragma directives according to functional category, which allows you to look up and link to pragmas by function; and includes individual descriptions of pragmas sorted alphabetically.

Chapter 5, “Compiler predefined macros,” on page 325 provides a list of compiler macros according to category.

Chapter 7, “Compiler built-in functions,” on page 339 contains individual descriptions of XL C/C++ built-in functions for Power architectures, categorized by their functionality.

Conventions

Typographical conventions

The following table explains the typographical conventions used in the IBM XL C/C++ for Multicore Acceleration for Linux, V10.1 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, xlc and xlc (xlc++), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
<i>italics</i>	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the <i>size</i> parameter if you return more than the <i>size</i> requested.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	nomaf <u>maf</u>

Table 1. *Typographical conventions (continued)*

Typeface	Indicates	Example
monospace	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.c, enter: xlc myprogram.c -O3.

Other conventions







In addition to typographical conventions, the following conventions are used:

- *(SPU only)* indicates functionality that only applies to code targeting the Synergistic Processor Unit (SPU), whether it is compiled using an spu or cbe prefixed compiler invocation command.
- *(PPU only)* indicates functionality that only applies to code targeting the Power Processor Unit (PPU), whether it is compiled using a ppu or cbe prefixed compiler invocation command.

Qualifying elements (icons)

Most features described in this information apply to both C and C++ languages. In descriptions of language elements where a feature is exclusive to one language, or where functionality differs between languages, this information uses icons to delineate segments of text as follows:

Table 2. *Qualifying elements*

Qualifier/Icon	Meaning
C only, or C only begins   C only ends	The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.
C++ only, or C++ only begins   C++ only ends	The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.
IBM extension begins   IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.

Syntax diagrams

Throughout this information, diagrams illustrate XL C/C++ syntax. This section will help you to interpret and use those diagrams.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

The ►— symbol indicates the beginning of a command, directive, or statement.

The —► symbol indicates that the command, directive, or statement syntax is continued on the next line.

The ►— symbol indicates that a command, directive, or statement is continued from the previous line.

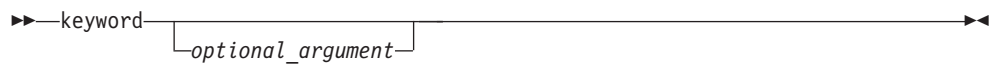
The —►◄ symbol indicates the end of a command, directive, or statement.

Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the |— symbol and end with the —| symbol.

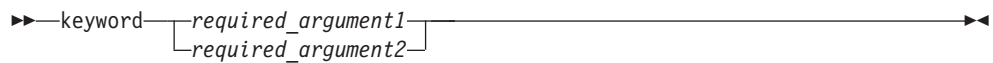
- Required items are shown on the horizontal line (the main path):



- Optional items are shown below the main path:



- If you can choose from two or more items, they are shown vertically, in a stack. If you *must* choose one of the items, one item of the stack is shown on the main path.



If choosing one of the items is optional, the entire stack is shown below the main path.



- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:



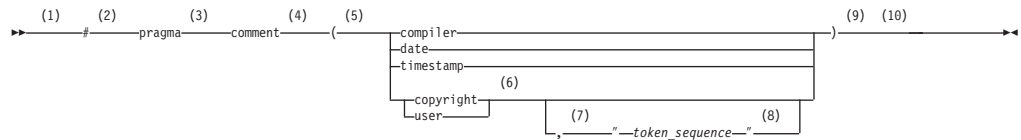
- The item that is the default is shown above the main path.



- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sample syntax diagram

The following syntax diagram example shows the syntax for the **#pragma comment** directive.



Notes:

- 1 This is the start of the syntax diagram.
- 2 The symbol # must appear first.
- 3 The keyword pragma must appear following the # symbol.
- 4 The name of the pragma comment must appear following the keyword pragma.
- 5 An opening parenthesis must be present.
- 6 The comment type must be entered only as one of the types indicated: compiler, date, timestamp, copyright, or user.
- 7 A comma must appear between the comment type copyright or user, and an optional character string.
- 8 A character string must follow the comma. The character string must be enclosed in double quotation marks.
- 9 A closing parenthesis is required.
- 10 This is the end of the syntax diagram.

The following examples of the **#pragma comment** directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

Examples in this information

The examples in this information, except where otherwise noted, are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible methods to achieve a specific result. Also, examples may use different compiler invocation commands interchangeably or simply indicate *invocation*. For detailed information on the commands available to invoke the compiler see "Invoking the compiler" in the *XL C/C++ Compiler Reference*.

The examples for installation information are labelled as either *Example* or *Basic example*. *Basic examples* are intended to document a procedure as it would be performed during a basic, or default, installation; these need little or no modification.

Related information

The following sections provide related information for XL C/C++:

IBM XL C/C++ information

XL C/C++ provides product information in the following formats:

- **README files**
README files contain late-breaking information, including changes and corrections to the product information. README files are located by default in the XL C/C++ directory and in the root directory of the installation CD.
- **Installable man pages**
Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL C/C++ for Multicore Acceleration for Linux, V10.1 Installation Guide*.
- **Information center**
The information center of searchable HTML files is viewable on the Web at <http://publib.boulder.ibm.com/infocenter/cellcomp/v101v121/index.jsp>.
- **PDF documents**
PDF documents are located by default in the `/opt/ibmcmp/xlc/cbe/10.1/doc/en_US/pdf/` directory. The PDF files are also available on the Web at <http://www.ibm.com/software/awdtools/xlcpp/multicore/library/>.
The following files comprise the full set of XL C/C++ product information:

Table 3. XL C/C++ PDF files

Document title	PDF file name	Description
<i>IBM XL C/C++ for Multicore Acceleration for Linux, V10.1 Installation Guide, GC23-8574-00</i>	install.pdf	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
<i>Getting Started with IBM XL C/C++ for Multicore Acceleration for Linux, V10.1, GC23-8572-00</i>	getstart.pdf	Contains an introduction to the XL C/C++ product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
<i>IBM XL C/C++ for Multicore Acceleration for Linux, V10.1 Compiler Reference, SC23-8570-00</i>	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions.
<i>IBM XL C/C++ for Multicore Acceleration for Linux, V10.1 Language Reference, SC23-8573-00</i>	langref.pdf	Contains information about the C and C++ programming languages, as supported by IBM, including language extensions for portability and conformance to nonproprietary standards.
<i>IBM XL C/C++ for Multicore Acceleration for Linux, V10.1 Optimization and Programming Guide, SC23-8571-00</i>	proguide.pdf	Contains information on advanced programming topics, such as application porting, library development, application optimization, and the XL C/C++ high-performance libraries.

To read a PDF file, use the Adobe® Reader. If you do not have the Adobe Reader, you can download it (subject to license terms) from the Adobe Web site at <http://www.adobe.com>.

More information related to XL C/C++ including redbooks, white papers, tutorials, and other articles, is available on the Web at:

Standards and specifications

XL C/C++ is designed to support the following standards and specifications. You can refer to these standards for precise definitions of some of the features found in this information.

- *Information Technology – Programming languages – C, ISO/IEC 9899:1990*, also known as C89.
- *Information Technology – Programming languages – C, ISO/IEC 9899:1999*, also known as C99.
- *Information Technology – Programming languages – C++, ISO/IEC 14882:1998*, also known as C++98.
- *Information Technology – Programming languages – C++, ISO/IEC 14882:2003(E)*, also known as *Standard C++*.
- *Information Technology – Programming languages – Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. This draft technical report has been accepted by the C standards committee, and is available at <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf>.
- *Draft Technical Report on C++ Library Extensions, ISO/IEC DTR 19768*. This draft technical report has been submitted to the C++ standards committee, and is available at <http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1836.pdf>.
- *Altivec Technology Programming Interface Manual*, Motorola Inc. This specification for vector data types, to support vector processing technology, is available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.

Other IBM information

- *IBM C/C++ Language Extensions for Cell Broadband Engine Architecture*, available at <http://publib.boulder.ibm.com/infocenter/systems/topic/eiccg/eiccgkickoff.htm>
- Specifications, white papers, and other technical information for the Cell Broadband Engine™ architecture are available at http://www.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine.
- The Cell Broadband Engine resource center, at <http://www.ibm.com/developerworks/power/cell/>, is the central repository for technical information, including articles, tutorials, programming guides, and educational resources.
- The IBM Systems Information Center, at <http://publib.boulder.ibm.com/infocenter/systems/index.jsp> is a resource for technical information about IBM systems, including the Cell Broadband Engine solution.

Other information

- *Using the GNU Compiler Collection* available at <http://gcc.gnu.org/onlinedocs>

Technical support

Additional technical support is available from the XL C/C++ Support page at <http://www.ibm.com/software/awdtools/xlcpp/support>. This page provides a portal with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send e-mail to compinfo@ca.ibm.com.

For the latest information about XL C/C++, visit the product information site at <http://www.ibm.com/software/awdtools/xlcpp>.

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this information or any other XL C/C++ information, send your comments by e-mail to compinfo@ca.ibm.com.

Be sure to include the name of the information, the part number of the information, the version of XL C/C++, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

Chapter 1. Compiling and linking applications

By default, when you invoke the XL C/C++ compiler, all of the following phases of translation are performed:

- preprocessing of program source
- compiling and assembling into object files
- linking into an executable

These different translation phases are actually performed by separate executables, which are referred to as compiler *components*. However, you can use compiler options to perform only certain phases, such as preprocessing, or assembling. You can then reinvoke the compiler to resume processing of the intermediate output to a final executable.

The following sections describe how to invoke the XL C/C++ compiler to preprocess, compile and link source files and libraries:

- “Invoking the compiler”
- “Types of input files” on page 4
- “Types of output files” on page 5
- “Specifying compiler options” on page 6
- “Reusing GNU C/C++ compiler options with **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++**” on page 10
- “Preprocessing” on page 12
- “Linking” on page 13
- “Compiler messages and listings” on page 15

Invoking the compiler

Different forms of the XL C/C++ compiler invocation commands support various levels of the C and C++ languages. In most cases, you should use the **xlc** command to compile your C source files, and the **xlc++** command to compile C++ source files. Use **xlc++** to link if you have both C and C++ object files. For applications targetting the Cell Broadband Engine processors, these commands are prefixed by **ppu**, **spu** or **cbe** depending on whether your program targets only the PPU, only the SPU, or contains both PPU and SPU sections.

You can use other forms of the command if your particular environment requires it. Table 4 on page 2 lists the different basic commands, with the “special” versions of each basic command. “Special” commands are described in Table 5 on page 3.

Note that for each invocation command, the compiler configuration file defines default option settings and, in some cases, macros; for information on the defaults implied by a particular invocation, see the `/opt/ibmcomp/xlc/cbe/10.1/etc/vac.cfg` file for your system.

Table 4. Compiler invocations

Basic invocations	Description	Equivalent special invocations
ppuxlc	Invokes the compiler for C source files and creates an executable that will run on the PPU. This command supports all of the ISO C99 standard features, and most IBM language extensions. This invocation is recommended for all applications targeting the PPU.	ppuxlc_r
spuxlc	Invokes the compiler for C source files and creates an executable that will run on the SPU. You must use this command for code you intend to run on the SPU.	
cbexlc	Invokes the compiler for C source files and creates an executable that will run on both the PPU and SPU. You must use this command for code in which the SPU portions have been written using the OpenMP directives.	
ppuc99	Invokes the compiler for C source files and creates an executable that will run on the PPU. This command supports all ANSI C89 features, and only the IBM language extensions related to vector processing. Use this invocation for strict conformance to the C99 standard.	ppuc99_r
ppuc89	Invokes the compiler for C source files and creates an executable that will run on the PPU. This command supports all ANSI C89 features, and only the IBM language extensions related to vector processing. Use this invocation for strict conformance to the C89 standard.	ppuc89_r
ppucc	Invokes the compiler for C source files and creates an executable that will run on the PPU. This command supports pre-ANSI C, and many common language extensions. You can use this command to compile legacy code that does not conform to standard C.	ppucc_r
ppugxlc	Invokes the compiler for C source files and creates an executable that will run on the PPU. This command accepts many common GNU C options, maps them to their XL C option equivalents, and then invokes ppuxlc . For more information, refer to “Reusing GNU C/C++ compiler options with ppugxlc , ppugxlc++ , spugxlc , and spugxlc++ ” on page 10.	
spugxlc	Invokes the compiler for C source files and creates an executable that will run on the SPU. This command accepts many common GNU C options, maps them to their XL C option equivalents, and then invokes spuxlc . For more information, refer to “Reusing GNU C/C++ compiler options with ppugxlc , ppugxlc++ , spugxlc , and spugxlc++ ” on page 10.	
ppuxlc++, ppuxlC	Invokes the compiler for C++ source files and creates an executable that will run on the PPU. If any of your source files are C++, you must use this invocation to link with the correct runtime libraries. Files with .c suffixes, assuming you have not used the --c compiler option, are compiled as C language source code.	ppuxlc++_r, ppuxlC_r

Table 4. Compiler invocations (continued)

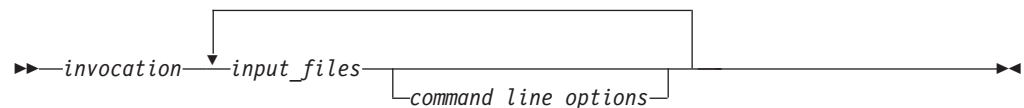
Basic invocations	Description	Equivalent special invocations
spuxlc++, spuxlC	<p>Invokes the compiler for C++ source files and creates an executable that will run on the SPU. If any of your source files are C++, you must use this invocation to link with the correct runtime libraries. You must use this command for code you intend to run on the SPU.</p> <p>Files with .c suffixes, assuming you have not used the <code>--</code> compiler option, are compiled as C language source code.</p>	
cbexlc++, cbexlC	<p>Invokes the compiler for C++ source files and creates an executable that will run on both the PPU and SPU. You must use this command for code in which the SPU portions have been written using the OpenMP directives. If any of your source files are written in C++, you must use this invocation to link with the correct runtime libraries.</p> <p>Files with .c suffixes, assuming you have not used the <code>--</code> compiler option, are compiled as C language source code.</p>	
ppugxlc++, ppugxlC	<p>Invokes the compiler for C++ files and creates an executable that will run on the PPU. This command accepts many common GNU C/C++ options, maps them to their XL C/C++ option equivalents, and then invokes ppuxlc++. For more information, refer to “Reusing GNU C/C++ compiler options with ppugxlc, ppugxlc++, spugxlc, and spugxlc++” on page 10.</p>	
spugxlc++, spugxlC	<p>Invokes the compiler for C++ files and creates an executable that will run on the SPU. This command accepts many common GNU C/C++ options, maps them to their XL C/C++ option equivalents, and then invokes spuxlc++. For more information, refer to “Reusing GNU C/C++ compiler options with ppugxlc, ppugxlc++, spugxlc, and spugxlc++” on page 10.</p>	

Table 5. Suffixes for special invocations

_r-suffixed invocations	All _r-suffixed invocations allow for threadsafe compilation and you can use them to link the programs that use multi-threading. Use these commands if you want to create threaded applications.
-------------------------	--

Command-line syntax

You invoke the compiler using the following syntax, where *invocation* can be replaced with any valid XL C/C++ invocation command listed in Table 4 on page 2:



The parameters of the compiler invocation command can be the names of input files, compiler options, and linker options.

Your program can consist of several input files. All of these source files can be compiled at once using only one invocation of the compiler. Although more than one source file can be compiled using a single invocation of the compiler, you can specify only one set of compiler options on the command line per invocation. Each distinct set of command-line compiler options that you want to specify requires a separate invocation.

Compiler options perform a wide variety of functions, such as setting compiler characteristics, describing the object code and compiler output to be produced, and performing some preprocessor functions.

By default, the invocation command calls *both* the compiler and the linker. It passes linker options to the linker. Consequently, the invocation commands also accept all linker options. To compile without linking, use the **-c** compiler option. The **-c** option stops the compiler after compilation is completed and produces as output, an object file *file_name.o* for each *file_name.nnn* input source file, unless you use the **-o** option to specify a different object file name. The linker is not invoked. You can link the object files later using the same invocation command, specifying the object files without the **-c** option.

Related information

- “Types of input files”

Types of input files

The compiler processes the source files in the order in which they appear. If the compiler cannot find a specified source file, it produces an error message and the compiler proceeds to the next specified file. However, the linker will not be run and temporary object files will be removed.

By default, the compiler preprocesses and compiles all the specified source files. Although you will usually want to use this default, you can use the compiler to preprocess the source file without compiling; see “Preprocessing” on page 12 for details.

You can input the following types of files to the XL C/C++ compiler:

C and C++ source files

These are files containing C or C++ source code.

To use the C compiler to compile a C language source file, the source file must have a **.c** (lowercase c) suffix, unless you compile with the **-qsource=c** option.

To use the C++ compiler, the source file must have a **.C** (uppercase C), **.cc**, **.cp**, **.cpp**, **.cxx**, or **.c++** suffix, unless you compile with the **-+** or **-qsource=c++** option.

Preprocessed source files

Preprocessed source files have a **.i** suffix, for example, *file_name.i*. The compiler sends the preprocessed source file, *file_name.i*, to the compiler where it is preprocessed again in the same way as a **.c** or **.C** file. Preprocessed files are useful for checking macros and preprocessor directives.

Object files

Object files must have a **.o** suffix, for example, *file_name.o*. Object files,

library files, and unstripped executable files serve as input to the linker. After compilation, the linker links all of the specified object files to create an executable file.

Assembler files

Assembler files must have a `.s` suffix, for example, `file_name.s`, unless you compile with the `-qsourcetype=assembler` option. Assembler files are assembled to create an object file.

Unpreprocessed assembler files

Unpreprocessed assembler files must have a `.S` suffix, for example, `file_name.S`, unless you compile with the `-qsourcetype=assembler-with-cpp` option. The compiler compiles all source files with a `.S` extension as if they are assembler language source files that need preprocessing.

Shared library files (PPU only)

Shared library files generally have a `.a` suffix, for example, `file_name.a`, but they can also have a `.so` suffix, for example, `file_name.so`.

Unstripped executable files

Executable and linking format (ELF) files that have not been stripped with the operating system `strip` command can be used as input to the compiler.

Related information

- Options summary by functional category: Input control

Types of output files

You can specify the following types of output files when invoking the XL C/C++ compiler:

Executable files

By default, executable files are named `a.out`. To name the executable file something else, use the `-o file_name` option with the invocation command. This option creates an executable file with the name you specify as `file_name`. The name you specify can be a relative or absolute path name for the executable file.

Object files

If you specify the `-c` option, an output object file, `file_name.o`, is produced for each input file. The linker is not invoked, and the object files are placed in your current directory. All processing stops at the completion of the compilation. The compiler gives object files a `.o` suffix, for example, `file_name.o`, unless you specify the `-o file_name` option, giving a different suffix or no suffix at all.

You can link the object files later into a single executable file by invoking the compiler.

Shared library files (PPU only)

If you specify the `-qmkshrobj` option, the compiler generates a single shared library file for all input files. The compiler names the output file `a.out`, unless you specify the `-o file_name` option, and give the file a `.so` suffix.

Assembler files

If you specify the `-S` option, an assembler file, `file_name.s`, is produced for each input file.

You can then assemble the assembler files into object files and link the object files by reinvoking the compiler.

Preprocessed source files

If you specify the **-P** option, a preprocessed source file, *file_name.i*, is produced for each input file.

You can then compile the preprocessed files into object files and link the object files by reinvoking the compiler.

Listing files

If you specify any of the listing-related options, such as **-qlist** or **-qsource**, a compiler listing file, *file_name.lst*, is produced for each input file. The listing file is placed in your current directory.

Target files

If you specify the **-M** or **-qmakedep** option, a target file suitable for inclusion in a makefile, *file_name.d* is produced for each input file.

Related information

- Options summary by functional category: Output control

Specifying compiler options

Compiler options perform a wide variety of functions, such as setting compiler characteristics, describing the object code and compiler output to be produced, and performing some preprocessor functions. You can specify compiler options in one or more of the following ways:

- On the command line
- In a custom configuration file, which is a file with a .cfg extension
- In your source program
- As system environment variables
- In a makefile

The compiler assumes default settings for most compiler options not explicitly set by you in the ways listed above.

When specifying compiler options, it is possible for option conflicts and incompatibilities to occur. XL C/C++ resolves most of these conflicts and incompatibilities in a consistent fashion, as follows:

In most cases, the compiler uses the following order in resolving conflicting or incompatible options:

1. Pragma statements in source code will override compiler options specified on the command line.
2. Compiler options specified on the command line will override compiler options specified as environment variables or in a configuration file. If conflicting or incompatible compiler options are specified in the same command line compiler invocation, the option appearing later in the invocation takes precedence.
3. Compiler options specified as environment variables will override compiler options specified in a configuration file.
4. Compiler options specified in a configuration file, command line or source program will override compiler default settings.

Option conflicts that do not follow this priority sequence are described in “Resolving conflicting compiler options” on page 9.

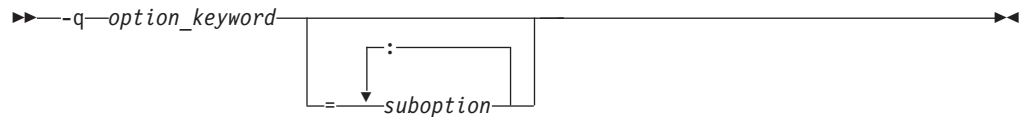
Specifying compiler options on the command line

Most options specified on the command line override both the default settings of the option and options set in the configuration file. Similarly, most options specified on the command line are in turn overridden by pragma directives, which provide you a means of setting compiler options right in the source file. Options that do not follow this scheme are listed in “Resolving conflicting compiler options” on page 9.

There are two kinds of command-line options:

- **-qoption_keyword** (compiler-specific)
- Flag options

-q options



Command-line options in the **-qoption_keyword** format are similar to on and off switches. For *most* **-q** options, if a given option is specified more than once, the last appearance of that option on the command line is the one recognized by the compiler. For example, **-qsource** turns on the source option to produce a compiler listing, and **-qnosource** turns off the source option so no source listing is produced. For example:

```
xlc -qnosource MyFirstProg.c -qsource MyNewProg.c
```

would produce a source listing for both `MyNewProg.c` and `MyFirstProg.c` because the last **source** option specified (**-qsource**) takes precedence.

You can have multiple **-qoption_keyword** instances in the same command line, but they must be separated by blanks. Option keywords can appear in either uppercase or lowercase, but you must specify the **-q** in lowercase. You can specify any **-qoption_keyword** before or after the file name. For example:

```
xlc -qLIST -qfloat=nomaf file.c
xlc file.c -qxref -qsource
```

You can also abbreviate many compiler options. For example, specifying **-qopt** is equivalent to specifying **-qoptimize** on the command line.

Some options have suboptions. You specify these with an equal sign following the **-qoption**. If the option permits more than one suboption, a colon (:) must separate each suboption from the next. For example:

```
xlc -qflag=w:e -qattr=full file.c
```

compiles the C source file `file.c` using the option **-qflag** to specify the severity level of messages to be reported. The **-qflag** suboption **w** (warning) sets the minimum level of severity to be reported on the listing, and suboption **e** (error) sets the minimum level of severity to be reported on the terminal. The **-qattr** with suboption **full** will produce an attribute listing of all identifiers in the program.

Flag options

XL C/C++ supports a number of common conventional flag options used on UNIX systems. Lowercase flags are different from their corresponding uppercase flags.

For example, `-c` and `-C` are two different compiler options: `-c` specifies that the compiler should only preprocess and compile and not invoke the linker, while `-C` can be used with `-P` or `-E` to specify that user comments should be preserved.

XL C/C++ also supports flags directed to other programming tools and utilities (for example, the `ld` command). The compiler passes on those flags directed to `ld` at link time.

Some flag options have arguments that form part of the flag. For example:

```
xlc stem.c -F/home/tools/test3/new.cfg:xlc
```

where `new.cfg` is a custom configuration file.

You can specify flags that do not take arguments in one string. For example:

```
xlc -Ocv file.c
```

has the same effect as:

```
xlc -O -c -v file.c
```

and compiles the C source file `file.c` with optimization (`-O`) and reports on compiler progress (`-v`), but does not invoke the linker (`-c`).

A flag option that takes arguments can be specified as part of a single string, but you can only use one flag that takes arguments, and it must be the last option specified. For example, you can use the `-o` flag (to specify a name for the executable file) together with other flags, only if the `-o` option and its argument are specified last. For example:

```
xlc -Ovo test test.c
```

has the same effect as:

```
xlc -O -v -otest test.c
```

Most flag options are a single letter, but some are two letters. Note that specifying `-pg` (extended profiling) is not the same as specifying `-p -g` (`-p` for profiling, and `-g` for generating debug information). Take care not to specify two or more options in a single string if there is another option that uses that letter combination.

Specifying compiler options in a configuration file

The default configuration file (`/opt/ibmcmp/xlc/cbe/10.1/etc/vac.cfg`) defines values and compiler options for the compiler. The compiler refers to this file when compiling C or C++ programs. The configuration file is a plain text file. You can edit this file, or create an additional customized configuration file to support specific compilation requirements. For more information, see “Using custom compiler configuration files” on page 24.

Specifying compiler options in program source files

You can specify compiler options within your program source by using pragma directives. A pragma is an implementation-defined instruction to the compiler. For those options that have equivalent pragma directives, there are several ways to specify the syntax of the pragmas:

- Using **#pragma options option_name** syntax — Many command-line options allow you to use the **#pragma options** syntax, which takes the same name as the option, and suboptions with a syntax identical to that of the option. For example, if the command-line option is:

```
-qhalt=w
```

The pragma form is:

```
#pragma options halt=w
```

The descriptions for each individual option indicates whether this form of the pragma is supported; also, for a complete list of these, see “#pragma options” on page 294.

- Using **#pragma name** syntax — Some options also have corresponding pragma directives that use a pragma-specific syntax, which may include additional or slightly different suboptions. Throughout the section “Individual option descriptions” on page 46, each option description indicates whether this form of the pragma is supported, and the syntax is provided.
- Using the standard C99 `_Pragma` operator — For options that support either forms of the pragma directives listed above, you can also use the C99 `_Pragma` operator syntax in both C and C++.

Complete details on pragma syntax are provided in “Pragma directive syntax” on page 269.

Other pragmas do not have equivalent command-line options; these are described in detail throughout Chapter 4, “Compiler pragmas reference,” on page 269.

Options specified with pragma directives in program source files override all other option settings, except other pragma directives. The effect of specifying the same pragma directive more than once varies. See the description for each pragma for specific information.

Pragma settings can carry over into included files. To avoid potential unwanted side effects from pragma settings, you should consider resetting pragma settings at the point in your program source where the pragma-defined behavior is no longer required. Some pragma options offer **reset** or **pop** suboptions to help you do this. These suboptions are listed in the detailed descriptions of the pragmas to which they apply.

Resolving conflicting compiler options

In general, if more than one variation of the same option is specified (with the exception of **-qxref** and **-qattr**), the compiler uses the setting of the last one specified. Compiler options specified on the command line must appear in the order you want the compiler to process them.

Two exceptions to the rules of conflicting options are the **-Idirectory** and **-Ldirectory** options, which have cumulative effects when they are specified more than once.

In most cases, the compiler uses the following order in resolving conflicting or incompatible options:

1. Pragma statements in source code override compiler options specified on the command line.
2. Compiler options specified on the command line override compiler options specified as environment variables or in a configuration file. If conflicting or

incompatible compiler options are specified on the command line, the option appearing later on the command line takes precedence.

3. Compiler options specified as environment variables override compiler options specified in a configuration file.
4. Compiler options specified in a configuration file override compiler default settings.

Not all option conflicts are resolved using the above rules. The table below summarizes exceptions and how the compiler handles conflicts between them.

Option	Conflicting options	Resolution
-qalias=allptrs	-qalias=noansi	-qalias=noansi
-qalias=typeptr	-qalias=noansi	-qalias=noansi
-qhalt	Multiple severities specified by -qhalt	Lowest severity specified
-qnoprint	-qxref, -qattr, -qsource, -qlistopt, -qlist	-qnoprint
-qfloat=rsqrt	-qnoignerrno	Last option specified
-qxref	-qxref=full	-qxref=full
-qattr	-qattr=full	-qattr=full
-qfloat=hsflt	-qfloat=spnans	-qfloat=hsflt
-E	-P, -o, -S	-E
-P	-c, -o, -S	-P
-#	-v	-#
-F	-B, -t, -W, -qpath	-B, -t, -W, -qpath
-qpath	-B, -t	-qpath
-S	-c	-S
-qnostdinc	-qc_stdinc, -qcpp_stdinc, -qgcc_c_stdinc, -qgcc_cpp_stdinc	-qnostdinc

Reusing GNU C/C++ compiler options with **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++**

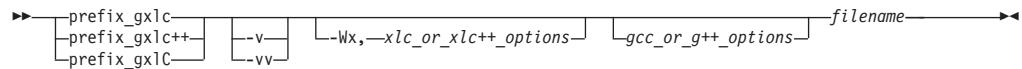
Each of the **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++** utilities accepts GNU C or C++ compiler options and translates them into comparable XL C/C++ options. Both utilities use the XL C/C++ options to create an **xlc** or **xlc++** invocation command, which they then use to invoke the compiler. These utilities are provided to facilitate the reuse of make files created for applications previously developed with GNU C/C++. However, to fully exploit the capabilities of XL C/C++, it is recommended that you use the XL C/C++ invocation commands and their associated options.

The actions of **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++** are controlled by the configuration file `/opt/ibmcmp/xlc/cbe/10.1/etc/gxlc.cfg`. The GNU C/C++ options that have an XL C or XL C++ counterpart are shown in this file. Not every GNU option has a corresponding XL C/C++ option. **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++** return warnings for input options that were not translated.

The **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++** option mappings are modifiable. For information on adding to or editing the **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++** configuration file, see “Configuring the **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** option mapping” on page 27.

ppugxlc, ppugxlc++, spugxlc, or spugxlc++ syntax

The following diagram shows the **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** syntax:



where:

prefix_ Can be *ppu* or *spu*.

filename

Is the name of the file to be compiled.

-v Allows you to verify the command that will be used to invoke XL C/C++. The utility displays the XL C/C++ invocation command that it has created, before using it to invoke the compiler.

-vv Allows you to run a simulation. The utility displays the XL C/C++ invocation command that it has created, but does not invoke the compiler.

-Wx,xlc_or_xlc++_options

Sends the given XL C/C++ options directly to the **ppuxlc**, **ppuxlc++**, **spuxlc**, or **spuxlc++** invocation command. The utility adds the given options to the XL C/C++ invocation it is creating, without attempting to translate them. Use this option with known XL C/C++ options to improve the performance of the utility. Multiple *xlc_or_xlc++_options* are delimited by a comma.

gcc_or_g++_options

The GNU C/C++ options that are to be translated to XL C/C++ options. The utility emits a warning for any option it cannot translate. The GNU C/C++ options that are currently recognized by **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** are in the configuration file *gxlc.cfg*. Multiple *gcc_or_g++_options* are delimited by the space character.

Example

To use the GCC **-fstrict-aliasing** option to compile the C version of the Hello World program, you can use:

```
ppugxlc -fstrict-aliasing hello.c
```

which translates into:

```
ppuxlc -qalias=ansi hello.c
```

This command is then used to invoke the XL C compiler.

Related information

- “Configuring the **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** option mapping” on page 27

Preprocessing

Preprocessing manipulates the text of a source file, usually as a first phase of translation that is initiated by a compiler invocation. Common tasks accomplished by preprocessing are macro substitution, testing for conditional compilation directives, and file inclusion.

You can invoke the preprocessor separately to process text without compiling. The output is an intermediate file, which can be input for subsequent translation. Preprocessing without compilation can be useful as a debugging aid because it provides a way to see the result of include directives, conditional compilation directives, and complex macro expansions.

The following table lists the options that direct the operation of the preprocessor.

Option	Description
"-E" on page 87	Preprocesses the source files and writes the output to standard output. By default, <code>#line</code> directives are generated.
"-P" on page 184	Preprocesses the source files and creates an intermediary file with a <code>.i</code> file name suffix for each source file. By default, <code>#line</code> directives are not generated.
"-qqpline" on page 191	Toggles on and off the generation of <code>#line</code> directives for the -E and -P options.
"-C, -C!" on page 66	Preserves comments in preprocessed output.
"-D" on page 80	Defines a macro name from the command line, as if in a <code>#define</code> directive.
"-U" on page 251	Undefines a macro name defined by the compiler or by the -D option.
"-qshowmacros" on page 212	Emits macro definitions to preprocessed output.

Directory search sequence for include files

XL C/C++ supports the following types of include files:

- Header files supplied by the compiler (referred to throughout this document as *XL C/C++ headers*)
- Header files mandated by the C and C++ standards (referred to throughout this document as *system headers*)
- Header files supplied by the operating system (also referred to throughout this document as *system headers*)
- User-defined header files


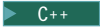


You can use any of the following methods to include any type of header file:

- Use the standard `#include <file_name>` preprocessor directive in the including source file.
- Use the standard `#include "file_name"` preprocessor directive in the including source file.
- Use the **-qinclude** compiler option.

If you specify the header file using a full (absolute) path name, you can use these methods interchangeably, regardless of the type of header file you want to include.

However, if you specify the header file using a *relative* path name, the compiler uses a different directory search order for locating the file depending on the method used to include the file.

Furthermore, the **-qidirfirst** and **-qstdinc** compiler options can affect this search order. The following summarizes the search order used by the compiler to locate header files depending on the mechanism used to include the files and on the compiler options that are in effect:

1. Header files included with **-qinclude** only: The compiler searches the current (working) directory from which the compiler is invoked.¹
2. Header files included with **-qinclude** or `#include "file_name"`: The compiler searches the directory in which the including file is located.¹
3. All header files: The compiler searches each directory specified by the **-I** compiler option, in the order that it appears on the command line.
4. All header files:  The compiler searches the standard directory for the XL C headers. The default directory for these headers is specified in the compiler configuration file. This is normally `/opt/ibmcmp/xlc/cbe/10.1/include/`, but the search path can be changed with the **-qc_stdinc** compiler option.  The compiler searches the standard directory for the XL C++ headers. The default directory for these headers is specified in the compiler configuration file. This is normally `/opt/ibmcmp/xlc/cbe/10.1//include/`, but the search path can be changed with the **-qcpp_stdinc** compiler option.²
5. All header files:  The compiler searches the standard directory for the system headers. The default directory for these headers is specified in the compiler configuration file. This is normally `/opt/ibmcmp/xlc/cbe/10.1/include/`, but the search path can be changed with the **-qgcc_c_stdinc** option.  The compiler searches the standard directory for the system headers. The default directory for these headers is specified in the compiler configuration file. This is normally `/opt/ibmcmp/xlc/cbe/10.1//include/` but the search path can be changed with the **-qgcc_cpp_stdinc** option.²

Note:

1. If the **-qidirfirst** compiler option is in effect, step 3 is performed before steps 1 and 2.
2. If the **-qnostdinc** compiler option is in effect, steps 4 and 5 are omitted.

Related information

- “-I” on page 118
- “-qc_stdinc (C only)” on page 78
- “-qcpp_stdinc (C++ only)” on page 79
- “-qgcc_c_stdinc (C only)” on page 109
- “-qgcc_cpp_stdinc (C++ only)” on page 110
- “-qidirfirst” on page 119
- “-qinclude” on page 122
- “-qstdinc” on page 225

Linking

The linker links specified object files to create one executable file. Invoking the compiler with one of the invocation commands automatically calls the linker unless you specify one of the following compiler options: **-E**, **-P**, **-c**, **-S**, **-qsyntaxonly** or **-#**.

Input files

Object files, unstripped executable files, and library files serve as input to the linker. Object files must have a `.o` suffix, for example, *filename.o*. Static library file names have an `.a` suffix, for example, *filename.a*. Dynamic library file names typically have a `.so` suffix, for example, *filename.so*.

Output files

The linker generates an *executable file* and places it in your current directory. The default name for an executable file is `a.out`. To name the executable file explicitly, use the `-o file_name` option with the compiler invocation command, where *file_name* is the name you want to give to the executable file. For example, to compile `myfile.c` and generate an executable file called `myfile`, enter:

```
ppuxlc myfile.c -o myfile
```

If you use the `-qmkshrobj` option to create a shared library, the default name of the shared object created is `a.out`. You can use the `-o` option to rename the file and give it a `.so` suffix.

You can invoke the linker explicitly with the `ppu-lb` or `spu-lb` command. However, the compiler invocation commands set several linker options, and link some standard files into the executable output by default. In most cases, it is better to use one of the compiler invocation commands to link your object files. For a complete list of options available for linking, see “Linking” on page 44.

Related information

“`-qmkshrobj` (PPU only)” on page 176

Order of linking

The compiler links libraries in the following order:

1. System startup libraries
2. User `.o` files and libraries
3. XL C/C++ libraries
4. C++ standard libraries
5. C standard libraries

Related information

- “Linking” on page 44

Redistributable libraries

If you build your application using XL C/C++, it may use one or more of the following redistributable libraries. If you ship the application, ensure that the users of the application have the packages containing the libraries. To make sure the required libraries are available to users, one of the following can be done:

- You can ship the packages that contain the redistributable libraries with the application. The packages are stored under the `rpms/` directory under the appropriate Linux distribution directory on the installation CD.
- The user can download the packages that contain the redistributable libraries from the XL C/C++ support Web site at:

<http://www.ibm.com/software/awdtools/xlcpp/support/>

For information on the licensing requirements related to the distribution of these packages refer to `LicAgree.pdf` on the CD.

Table 6. Redistributable libraries

Package name	Libraries (and default installation path)	Description
cell-xlc-rte	/opt/ibmcmp/lib/cbe/libibmc++.so.1 /opt/ibmcmp/lib64/cbe/libibmc++.so.1	XL C++ runtime libraries


Compiler messages and listings

The following sections discuss the various methods of reporting provided by the compiler after compilation.

- “Compiler messages”
- “Compiler return codes” on page 17
- “Compiler listings” on page 18

Compiler messages

When the compiler encounters a programming error while compiling a C or C++ source program, it issues a diagnostic message to the standard error device and, if you compile with the **-qsource** option, to a listing file. Note that messages are specific to the C or C++ language.

 If you specify the compiler option **-qsrcmsg** and the error is applicable to a particular line of code, the reconstructed source line or partial source line is included with the error message. A reconstructed source line is a preprocessed source line that has all the macros expanded.

You can control the diagnostic messages issued, according to their severity, using either the **-qflag** option or the **-w** option. To get additional informational messages about potential problems in your program, use the **-qinfo** option.

Related information

- “-qsource” on page 218
- “-qsrcmsg (C only)” on page 221
- “-qflag” on page 97
- “-w” on page 261
- “-qinfo” on page 124

Compiler message format

Diagnostic messages have the following format:

`"file", line line_number.column_number: 15dd-number (severity) text.`

where:

file Is the name of the C or C++ source file with the error.

line_number

Is the source code line number where the error was found.

column_number

Is the source code column number where the error was found.

15 Is the compiler product identifier.

dd is a two-digit code indicating the compiler component that issued the message. *dd* can have the following values:

00 - code generating or optimizing message

- 01 - compiler services message
- 05 - message specific to the C compiler
- 06 - message specific to the C compiler
- 40 - message specific to the C++ compiler
- 86 - message specific to interprocedural analysis (IPA)

number


Is the message number.

severity

Is a letter representing the severity of the error. See “Message severity levels and compiler response” for a description of these.

text

Is a message describing the error.

 If you compile with **-qsrcmsg**, diagnostic messages have the following format:

x - 15dd-*nnn*(*severity*) *text*.

where *x* is a letter referring to a finger in the finger line.

Message severity levels and compiler response

XL C/C++ uses a multi-level classification scheme for diagnostic messages. Each level of severity is associated with a compiler response. The following table provides a key to the abbreviations for the severity levels and the associated default compiler response. Note that you can adjust the default compiler response by using any of the following options:


- **-qhalt** allows you to halt the compilation phase at a lower severity level than the default
- **-qmaxerr** allows you to halt the compilation phase as soon as a specific number of errors at a specific severity level is reached
-  **-qhaltonmsg** allows you to halt the compilation phase as soon as a specific error is encountered

Table 7. Compiler message severity levels



Letter	Severity	Compiler response
I	Informational	Compilation continues and object code is generated. The message reports conditions found during compilation.
W	Warning	Compilation continues and object code is generated. The message reports valid but possibly unintended conditions.
 E	Error	Compilation continues and object code is generated. Error conditions exist that the compiler can correct, but the program might not produce the expected results.

Table 7. Compiler message severity levels (continued)


Letter	Severity	Compiler response
S	Severe error	<p>Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct:</p> <ul style="list-style-type: none"> • If the message indicates a resource limit (for example, file system full or paging space full), provide additional resources and recompile. • If the message indicates that different compiler options are needed, recompile using them. • Check for and correct any other errors reported prior to the severe error. • If the message indicates an internal compiler error, the message should be reported to your IBM service representative.
 C++ U	Unrecoverable error	<p>The compiler halts. An internal compiler error has occurred. The message should be reported to your IBM service representative.</p>

Related information

- “-qhalt” on page 113
- “-qmaxerr” on page 170
- “-qhaltonmsg (C++ only)” on page 114
- Options summary by functional category: Listings and messages

Compiler return codes

At the end of compilation, the compiler sets the return code to zero under any of the following conditions:

- No messages are issued.
- The highest severity level of all errors diagnosed is less than the setting of the **-qhalt** compiler option, and the number of errors did not reach the limit set by the **-qmaxerr** compiler option.
-  C++ No message specified by the **-qhaltonmsg** compiler option is issued.

Otherwise, the compiler sets the return code to one of the following values:

Return code	Error type
1	Any error with a severity level higher than the setting of the -qhalt compiler option has been detected.
40	An option error or an unrecoverable error has been detected.
41	A configuration file error has been detected.
249	A no-files-specified error has been detected.
250	An out-of-memory error has been detected. The compiler cannot allocate any more memory for its use.
251	A signal-received error has been detected. That is, an unrecoverable error or interrupt signal has occurred.
252	A file-not-found error has been detected.
253	An input/output error has been detected: files cannot be read or written to.
254	A fork error has been detected. A new process cannot be created.

255	An error has been detected while the process was running.
-----	---

Note: Return codes may also be displayed for runtime errors.

ppugxlc, ppugxlc++, spugxlc, and spugxlc++ return codes

Like other invocation commands, **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++** return output, such as listings, diagnostic messages related to the compilation, warnings related to unsuccessful translation of GNU options, and return codes. If **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** cannot successfully call the compiler, it sets the return code to one of the following values:

- 40** A **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** option error or unrecoverable error has been detected.
- 255** An error has been detected while the process was running.

Compiler listings

A listing is a compiler output file (with a .lst suffix) that contains information about a particular compilation. As a debugging aid, a compiler listing is useful for determining what has gone wrong in a compilation. For example, any diagnostic messages emitted during compilation are written to the listing.

To produce a listing, you can compile with any of the following options, which provide different types of information:

- **-qsource**
- **-qlistopt**
- **-qattr**
- **-qxref**
- **-qlist**

When any of these options is in effect, a listing file *filename.lst* is saved in the current directory for every input file named in the compilation.

Listing information is organized in sections. A listing contains a header section and a combination of other sections, depending on other options in effect. The contents of these sections are described as follows.

Header section

Lists the compiler name, version, and release, as well as the source file name and the date and time of the compilation.

Source section

If you use the **-qsource** option, lists the input source code with line numbers. If there is an error at a line, the associated error message appears after the source line. Lines containing macros have additional lines showing the macro expansion. By default, this section only lists the main source file. Use the **-qshowinc** option to expand all header files as well.

Options section

Lists the non-default options that were in effect during the compilation. To list all options in effect, specify the **-qlistopt** option.

Attribute and cross-reference listing section

If you use the **-qattr** or **-qxref** options, provides information about the variables used in the compilation unit, such as type, storage duration, scope, and where they are defined and referenced. Each of these options provides different information on the identifiers used in the compilation.

File table section

Lists the file name and number for each main source file and include file. Each file is associated with a file number, starting with the main source file, which is assigned file number 0. For each file, the listing shows from which file and line the file was included. If the **-qshowinc** option is also in effect, each source line in the source section will have a file number to indicate which file the line came from.

Compilation epilogue section

Displays a summary of the diagnostic messages by severity level, the number of source lines read, and whether or not the compilation was successful.

Object or assembly section

If you use the **-qlist** option, lists the object or assembly code generated by the compiler for the PPU or lists the object code generated by the compiler for the SPU. This section is useful for diagnosing execution time problems, if you suspect the program is not performing as expected due to code generation error.

Related information

- Summary of command line options: Listings and messages

Chapter 2. Configuring compiler defaults

When you compile an application with XL C/C++, the compiler uses default settings that are determined in a number of ways:

- Internally defined settings. These settings are predefined by the compiler and you cannot change them.
- Settings defined by system environment variables. Certain environment variables are required by the compiler; others are optional. You may have already set some of the basic environment variables during the installation process (for more information, see the XL C/C++ Installation Guide). “Setting environment variables” provides a complete list of the required and optional environment variables you can set or reset after installing the compiler.
- Settings defined in the compiler configuration file, `vac.cfg`. The compiler requires many settings that are determined by its configuration file. Normally, the configuration file is automatically generated during the installation procedure. (For more information, see the XL C/C++ Installation Guide). However, you can customize this file after installation, to specify additional compiler options, default option settings, library search paths, and so on. Information on customizing the configuration file is provided in “Using custom compiler configuration files” on page 24.
- Settings defined by the GCC options configuration file. If you are using the **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** utility to map GCC options, the default option mappings are defined in the `/opt/ibmcmp/xlc/cbe/10.1/etc/gxlc.cfg` file. You can customize this file to suit your requirements; for more information, see “Configuring the **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** option mapping” on page 27.

Setting environment variables

To set environment variables in Bourne, Korn, and BASH shells, use the following commands:

```
variable=value  
export variable
```

where *variable* is the name of the environment variable, and *value* is the value you assign to the variable.

To set environment variables in the C shell, use the following command:

```
setenv variable value
```

where *variable* is the name of the environment variable, and *value* is the value you assign to the variable.

To set the variables so that all users have access to them, in Bourne, Korn, and BASH shells, add the commands to the file `/etc/profile`. To set them for a specific user only, add the commands to the file `.profile` in the user's home directory. In C shell, add the commands to the file `/etc/csh.cshrc`. To set them for a specific user only, add the commands to the file `.cshrc` in the user's home directory. The environment variables are set each time the user logs in.

The following sections discuss the environment variables you can set for XL C/C++ and applications you have compiled with it:

- “Compile-time and link-time environment variables”
- “Runtime environment variables”

Compile-time and link-time environment variables

The following environment variables are used by the compiler when you are compiling and linking your code. Many are built into the Linux operating system. All of these variables are optional.

LD_RUN_PATH

Specifies search paths for dynamically loaded libraries, equivalent to using the **-R** link-time option. The shared-library locations named by the environment variable are embedded into the executable, so the dynamic linker can locate the libraries at application run time. For more information about this environment variable, see your operating system documentation. See also “-R” on page 201.

PATH Specifies the directory search path for the executable files of the compiler. Executables are in `/opt/ibmcmp/xlc/cbe/10.1/bin/` if installed to the default location. For information, see *Setting the PATH environment variable to include the path to the XL C/C++ invocations* in the *XL C/C++ Installation Guide*

TMPDIR

Optionally specifies the directory in which temporary files are created during compilation. The default location, `/tmp/`, may be inadequate at high levels of optimization, where paging and temporary files can require significant amounts of disk space, so you can use this environment variable to specify an alternate directory.

XLC_USR_CONFIG

Specifies the location of a custom configuration file to be used by the compiler. The file name must be given with its absolute path. The compiler will first process the definitions in this file before processing those in the default system configuration file, or those in a customized file specified by the **-F** option; for more information, see “Using custom compiler configuration files” on page 24.

Runtime environment variables

The following environment variables are used by the system loader or by your application when it is executed. All of these variables are optional.

LD_LIBRARY_PATH

Specifies an alternate directory search path for dynamically linked libraries at application run time. If shared libraries required by your application have been moved to an alternate directory that was not specified at link time, and you do not want to relink the executable, you can set this environment variable to allow the dynamic linker to locate them at run time. For more information about this environment variable, see your operating system documentation.

PDFDIR

Optionally specifies the directory in which profiling information is saved when you run an application that you have compiled with the **-qpdf1** option. The default value is unset, and the compiler places the profile data file in the current working directory. When you recompile or relink your application with **-qpdf2**, the compiler uses the data saved in this directory to optimize the application. It is recommended that you set this variable to

an absolute path if you will be using profile-directed feedback. See “-qpdf1, -qpdf2 (PPU only)” on page 187 for more information.

Environment variables for parallel processing

If you are using OpenMP constructs for parallelization, you can also specify runtime options using OMP environment variables, as discussed in “OpenMP environment variables for parallel processing.”

Note: You must use a threadsafe compiler mode invocations when compiling parallelized program code. For applications targeting Cell Broadband Engine processors, you must use a single-source invocation such as **cbexlc**, **cbexlc++** or **cbexlC**.

Related information

- “Pragma directives for parallel processing” on page 312
- “Built-in functions for parallel processing” on page 366

OpenMP environment variables for parallel processing: OpenMP runtime options affecting parallel processing are set by specifying OMP environment variables. These environment variables use syntax of the form:

►►—*env_variable*—=*option_and_args*—►►

If an OMP environment variable is not explicitly set, its default setting is used.

For information on the OpenMP specification, see: www.openmp.org/specs.

OMP_SCHEDULE=algorithm environment variable: The **OMP_SCHEDULE** environment variable specifies the scheduling algorithm used for loops not explicitly assigned a scheduling algorithm with the **omp schedule** directive.

Valid options for *algorithm* are:

- runtime
- static[, *n*]

If specifying a chunk size with *n*, the value of *n* must be an integer value of 1 or greater.

The default scheduling algorithm is **static**.

OMP_NESTED: The **OMP_NESTED=TRUE|FALSE** environment variable enables or disables nested parallelism. Its setting can be overridden by calling the **omp_set_nested** runtime library function.

If nested parallelism is disabled, nested parallel regions are serialized and run in the current thread.

In the current implementation, nested parallel regions are always serialized. As a result, **OMP_SET_NESTED** does not have any effect, and **omp_get_nested** always returns 0. If **-qsmp=nested_par** option is on (only in non-strict OMP mode), nested parallel regions may employ additional threads as available. However, no new team will be created to run nested parallel regions.

The default value for **OMP_NESTED** is **FALSE**.

OMP_DYNAMIC=TRUE | FALSE environment variable: The **OMP_DYNAMIC** environment variable enables or disables dynamic adjustment of the number of threads available for running parallel regions.

If set to **TRUE**, it is ignored.

If set to **FALSE**, dynamic adjustment is disabled. This is the default.

OMP_NUM_THREADS: The **OMP_NUM_THREADS=num** environment variable gives you full control over the number of user threads used to run your program.

num represents the number of parallel threads requested, which is usually equivalent to the number of processors available on the system. This number can be overridden by calling the **omp_set_num_threads** runtime library function. The maximum number of threads is limited by the number of available SPUs.

The default value for *num* is the number of processors available on the system.

You can override the setting of **OMP_NUM_THREADS** for a given parallel section by using the **num_threads** clause available in several **#pragma omp** directives.

Using custom compiler configuration files

XL C/C++ generates a default configuration file `/opt/ibmcmp/xlc/cbe/10.1/etc/vac.cfg` at installation time. (See the *XL C/C++ Installation Guide* for more information on the various tools you can use to generate the configuration file during installation.) The configuration file specifies information that the compiler uses when you invoke it.

If you are running on a single-user system, or if you already have a compilation environment with compilation scripts or makefiles, you may want to leave the default configuration file as it is.

Otherwise, especially if you want many users to be able to choose among several sets of compiler options, you may want to use custom configuration files for specific needs. For example, you might want to enable **-qlist** by default for compilations using the **ppuxlc** compiler invocation command. Rather than a user being required to specify this option on the command line for every compilation, it would automatically be in effect every time the compiler is invoked with the **ppuxlc** command.

You have several options for customizing configuration files:

- You can directly edit the default configuration file. In this case, the customized options will apply for all users for all compilations. The disadvantage of this option is that you will need to reapply your customizations to the new default configuration file that is provided every time you install a compiler update.
- You can use the default configuration file as the basis of customized copies that you specify at compile time with the **-F** option. In this case, the custom file overrides the default file on a per-compilation basis. Again, the disadvantage of this option is that you will need to reapply your customizations to the new default configuration file that is provided every time you install a compiler update.
- You can create custom, or user-defined, configuration files that are specified at compile time with the **XLC_USR_CONFIG** environment variable. In this case, the custom user-defined files complement, rather than override, the default

configuration file, and they can be specified on a per-compilation or global basis. The advantage of this option is that you do not need to modify your existing, custom configuration files when a new system configuration file is installed during an update installation. Procedures for creating custom, user-defined configuration files are provided below.

Related information:

- “-F” on page 94
- “Compile-time and link-time environment variables” on page 22

Creating custom configuration files

If you use the `XLC_USR_CONFIG` environment variable to instruct the compiler to use a custom user-defined configuration file, the compiler will examine and process the settings in that user-defined configuration file before looking at the settings in the default system configuration file.

To create a custom user-defined configuration file, you add stanzas which specify multiple levels of the **use** attribute. The user-defined configuration file can reference definitions specified elsewhere in the same file, as well as those specified in the system configuration file. For a given compilation, when the compiler looks for a given stanza, it searches from the beginning of the user-defined configuration file and then follows any other stanza named in the **use** attribute, including those specified in the system configuration file.

If the stanza named in the **use** attribute has a name different from the stanza currently being processed, then the search for the **use** stanza starts from the beginning of the user-defined configuration file. This is the case for stanzas A, C, and D in the example shown below. However, if the stanza in the **use** attribute has the same name as the stanza currently being processed, as is the case of the two B stanzas in the example, then the search for the **use** stanza starts from the location of the current stanza.

The following example shows how you can use multiple levels for the **use** attribute. This example uses the **options** attribute to help show how the **use** attribute works, but any other attribute, such as **libraries** could also be used.

```
A: use =DEFLT
    options=<set of options A>
B: use =B
    options=<set of options B1>
B: use =D
    options=<set of options B2>
C: use =A
    options=<set of options C>
D: use =A
    options=<set of options D>
DEFLT:
    options=<set of options Z>
```

Figure 1. Sample configuration file

In this example:

- stanza A uses option sets A and Z
- stanza B uses option sets B1, B2, D, A, and Z
- stanza C uses option sets C, A, and Z

- stanza D uses option sets *D*, *A*, and *Z*

Attributes are processed in the same order as the stanzas. The order in which the options are specified is important for option resolution. Ordinarily, if an option is specified more than once, the last specified instance of that option wins.

By default, values defined in a stanza in a configuration file are added to the list of values specified in previously processed stanzas. For example, assume that the `XLC_USR_CONFIG` environment variable is set to point to the user-defined configuration file at `~/userconfig1`. With the user-defined and default configuration files shown in the example below, the compiler will reference the **ppuxlc** stanza in the user-defined configuration file and use the option sets specified in the configuration files in the following order: *A1*, *A*, *D*, and *C*.

```
ppuxlc: use=ppuxlc
       options= <A1>

DEFLT: use=DEFLT
       options=<D>
```

Figure 2. Custom user-defined configuration file `~/userconfig1`

```
ppuxlc: use=DEFLT
       options=<A>

DEFLT:
       options=<C>
```

Figure 3. Default configuration file `vac.cfg`

Overriding the default order of attribute values

You can override the default order of attribute values by changing the assignment operator(=) for any attribute in the configuration file.

Table 8. Assignment operators and attribute ordering

Assignment Operator	Description
--	Prepend the following values before any values determined by the default search order.
:=	Replace any values determined by the default search order with the following values.
+=	Append the following values after any values determined by the default search order.

For example, assume that the `XLC_USR_CONFIG` environment variable is set to point to the custom user-defined configuration file at `~/userconfig2`.

Custom user-defined configuration file

`~/userconfig2`

```
ppuxlc_prepend: use=ppuxlc
                options--<B1>
ppuxlc_replace: use=ppuxlc
                options:=<B2>
ppuxlc_append: use=ppuxlc
                options+=<B3>
```

```
DEFLT: use=DEFLT
       options=<D>
```

Default configuration file `vac.cfg`

```
ppuxlc: use=DEFLT
       options=<B>

DEFLT:
       options=<C>
```

The stanzas in the configuration files shown above will use the following option sets, in the following orders:

1. stanza `ppuxlc` uses *B*, *D*, and *C*
2. stanza `ppuxlc_prepend` uses *B1*, *B*, *D*, and *C*
3. stanza `ppuxlc_replace` uses *B2*
4. stanza `ppuxlc_append` uses *B*, *D*, *C*, and *B3*

You can also use assignment operators to specify an attribute more than once. For example:

```
ppuxlc:
  use=ppuxlc
  options=-Isome_include_path
  options+=some options
```

Figure 4. Using additional assignment operations

Examples of stanzas in custom configuration files

<pre>DEFLT: use=DEFLT options = -g</pre>	<p>This example specifies that the -g option is to be used in all compilations.</p>
<pre>ppuxlc: use=ppuxlc options+=-qlist ppuxlc_r: use=ppuxlc_r options+=-qlist</pre>	<p>This example specifies that -qlist be used for any compilation invoked by the ppuxlc and ppuxlc_r commands. This -qlist specification overrides the default setting of -qlist specified in the system configuration file.</p>
<pre>DEFLT: use=DEFLT libraries=-L/home/user/lib,-lmylib</pre>	<p>This example specifies that all compilations should link with <code>/home/user/lib/libmylib.a</code>.</p>

Configuring the `ppugxlc`, `ppugxlc++`, `spugxlc`, or `spugxlc++` option mapping

The `ppugxlc`, `ppugxlc++`, `spugxlc`, and `spugxlc++` utilities use the configuration file `/opt/ibmcmp/xlc/cbe/10.1/etc/gxlc.cfg` to translate GNU C and C++ options to XL C/C++ options. Each entry in `gxlc.cfg` describes how the utility should map a GNU C or C++ option to an XL C/C++ option and how to process it.

An entry consists of a string of flags for the processing instructions, a string for the GNU C/C++ option, and a string for the XL C/C++ option. The three fields must be separated by white space. If an entry contains only the first two fields and the XL C/C++ option string is omitted, the GNU C option in the second field will be recognized by `ppugxlc`, `ppugxlc++`, `spugxlc`, or `spugxlc++` and silently ignored.

The `#` character is used to insert comments in the configuration file. A comment can be placed on its own line, or at the end of an entry.

The following syntax is used for an entry in `gxlc.cfg`:

```
abcd    "gcc_or_g++_option"    "xlc_or_xlc++_option"
```

where:

- a* Lets you disable the option by adding **no-** as a prefix. The value is either **y** for yes, or **n** for no. For example, if the flag is set to **y**, then **inline** can be disabled as **fno-inline**, and the entry is:

yyn* "-finline" "-qinline"

If given **-fno-inline**, then the utility will translate it to **-qnoinline**.

- b* Informs the utility that the XL C/C++ option has an associated value. The value is either **y** for yes, or **n** for no. For example, if option **-fmyvalue=n** maps to **-qmyvalue=n**, then the flag is set to *y*, and the entry is:

nyn* "-fmyvalue" "-qmyvalue"

The utility will then expect a value for these options.

- c* Controls the processing of the options. The value can be any of the following:

- n** Tells the utility to process the option listed in the *gcc_or_g++_option* field
- i** Tells the utility to ignore the option listed in the *gcc_or_g++_option* field. The utility will generate a message that this has been done, and continue processing the given options.
- e** Tells the utility to halt processing if the option listed in the *gcc_or_g++_option* field is encountered. The utility will also generate an error message.

For example, the GCC option **-I-** is not supported and must be ignored by **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++**. In this case, the flag is set to *i*, and the entry is:

nni* "-I-"

If the utility encounters this option as input, it will not process it and will generate a warning.

- d* Lets **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** include or ignore an option based on the type of compiler. The value can be any of the following:

- c** Tells the utility to translate the option only for C.
- x** Tells the utility to translate the option only for C++.
- *** Tells **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** to translate the option for C and C++.

For example, **-fwritable-strings** is supported by both compilers, and maps to **-qnoro**. The entry is:

nnn* "-fwritable-strings" "-qnoro"

"gcc_or_g++_option"

Is a string representing a GNU C/C++ option. This field is required and must appear in double quotation marks.

"xlc_or_xlc++_option"

Is a string representing an XL C/C++ option. This field is optional, and, if present, must appear in double quotation marks. If left blank, the utility ignores the *gcc_or_g++_option* in that entry.

It is possible to create an entry that will map a range of options. This is accomplished by using the asterisk (*) as a wildcard. For example, the GCC **-D** option requires a user-defined name and can take an optional value. It is possible to have the following series of options:

```
-DCOUNT1=100
-DCOUNT2=200
-DCOUNT3=300
-DCOUNT4=400
```

Instead of creating an entry for each version of this option, the single entry is:

```
nnn*      "-D*"      "-D*"
```

where the asterisk will be replaced by any string following the **-D** option.

Conversely, you can use the asterisk to exclude a range of options. For example, if you want **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** to ignore all the **-std** options, then the entry would be:

```
nni*      "-std*"
```

When the asterisk is used in an option definition, option flags *a* and *b* are not applicable to these entries.

The character % is used with a GNU C/C++ option to signify that the option has associated parameters. This is used to insure that **ppugxlc**, **ppugxlc++**, **spugxlc**, or **spugxlc++** will ignore the parameters associated with an option that is ignored. For example, the **-isystem** option is not supported and uses a parameter. Both must be ignored by the application. In this case, the entry is:

```
nni*      "-isystem %"
```

For a complete list of GNU C and C++ and XL C/C++ option mappings, refer to:

<http://www.ibm.com/support/docview.wss?rs=2239&context=SSJT9L&uid=swg27010369>

Related information

- The GNU Compiler Collection online documentation at <http://gcc.gnu.org/onlinedocs/>

Chapter 3. Compiler options reference

The following sections contain a summary of the compiler options available in XL C/C++ by functional category, followed by detailed descriptions of the individual options.

Related information

- “Specifying compiler options” on page 6
- “Reusing GNU C/C++ compiler options with **ppugxlc**, **ppugxlc++**, **spugxlc**, and **spugxlc++**” on page 10

Summary of compiler options by functional category

The XL C/C++ options available are grouped into the following categories. If the option supports an equivalent pragma directive, this is indicated. To get detailed information on any option listed, see the full description for that option.

- “Output control”
- “Input control” on page 32
- “Language element control” on page 33
- “Template control (C++ only)” on page 34
- “Floating-point and integer control” on page 35
- “Error checking and debugging” on page 37
- “Listings, messages, and compiler information” on page 39
- “Optimization and tuning” on page 41
- “Object code control” on page 35
- “Linking” on page 44
- “Portability and migration” on page 45
- “Compiler customization” on page 45

Output control

The options in this category control the type of file output the compiler produces, as well as the locations of the output. These are the basic options that determine the compiler components that will be invoked; the preprocessing, compilation, and linking steps that will (or will not) be taken; and the kind of output to be generated.

Table 9. Compiler output options

Option name	Equivalent pragma name	Description
“-c” on page 65	None.	Prevents the completed object from being sent to the linker. With this option, the output is a .o file for each source file.
“-C, -C!” on page 66	None.	When used in conjunction with the -E or -P options, preserves or removes comments in preprocessed output.
“-E” on page 87	None.	Preprocesses the source files named in the compiler invocation, without compiling, and writes the output to the standard output.

Table 9. Compiler output options (continued)

Option name	Equivalent pragma name	Description
"-qmakedep, -M" on page 168	None.	Creates an output file containing targets suitable for inclusion in a description file for the make command.
"-MF" on page 174	None.	Specifies the target for the output generated by the -qmakedep or -M options.
"-qmkshrobj (PPU only)" on page 176	None.	Creates a shared object from generated object files.
"-o" on page 178	None.	Specifies a name for the output object, assembler, or executable file.
"-P" on page 184	None.	Preprocesses the source files named in the compiler invocation, without compiling, and creates an output preprocessed file for each input file.
"-S" on page 208	None.	Generates an assembler language file for each source file.
"-qshowmacros" on page 212	None.	Emits macro definitions to preprocessed output.
"-qtimestamps" on page 245	None.	Controls whether or not implicit time stamps are inserted into an object file.

Input control

The options in this category specify the type and location of your source files.

Table 10. Compiler input options

Option name	Equivalent pragma name	Description
"-+ (plus sign) (C++ only)" on page 47	None.	Compiles any file as a C++ language file.
"-qcinc (C++ only)" on page 71	None.	Places an extern "C" { } wrapper around the contents of include files located in a specified directory.
"-I" on page 118	None.	Adds a directory to the search path for include files.
"-qidirfirst" on page 119	#pragma options idirfirst	Specifies whether the compiler searches for user include files in directories specified by the -I option <i>before</i> or <i>after</i> searching any other directories.
"-qinclude" on page 122	None.	Specifies additional header files to be included in a compilation unit, as though the files were named in an #include statement in the source file.

Table 10. Compiler input options (continued)

Option name	Equivalent pragma name	Description
"-qsourcectype" on page 219	None.	Instructs the compiler to treat all recognized source files as a specified source type, regardless of the actual file name suffix.
"-qstdinc" on page 225	#pragma options stdinc	Specifies whether the standard include directories are included in the search paths for system and user header files.


Language element control

The options in this category allow you to specify the characteristics of the source code. You can also use these options to enforce or relax language restrictions, and enable or disable language extensions.

Table 11. Language element control options

Option name	Equivalent pragma name	Description
"-qaltivec" on page 56	None	Enables compiler support for vector data types and operators.
"-qasm" on page 58	None	Controls the interpretation of and subsequent generation of code for assembler language extensions.
"-qcpluscmt (C only)" on page 76	None.	Enables recognition of C++-style comments in C source files.
"-D" on page 80	None.	Defines a macro as in a #define preprocessor directive.
"-qdigraph" on page 83	#pragma options digraph	Enables recognition of digraph key combinations or keywords to represent characters not found on some keyboards.
"-qdollar" on page 85	#pragma options dollar	Allows the dollar-sign (\$) symbol to be used in the names of identifiers.
"-qea32, -qea64 (SPU only)" on page 88	None.	Compiles SPU code that uses effective address space support for either a 32-bit or 64-bit PPE address space.
"-qignprag" on page 121	#pragma options ignprag	Instructs the compiler to ignore certain pragma statements.
"-qkeyword" on page 145	None.	Controls whether the specified name is treated as a keyword or as an identifier whenever it appears in your program source.

Table 11. Language element control options (continued)

Option name	Equivalent pragma name	Description
"-qlanglvl" on page 148	 #pragma options langlvl, #pragma langlvl	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.
"-qlonglong" on page 167	#pragma options long long	Allows IBM long long integer types in your program.
"-qmbcs, -qdbcs" on page 173	#pragma options mbcs, #pragma options dbcs	Enables support for multibyte character sets (MBCS) and Unicode characters in your source code.
"-qstaticinline (C++ only)" on page 222	None.	Controls whether inline functions are treated as having static or extern linkage.
"-qtabsize" on page 237	None.	Sets the default tab length, for the purposes of reporting the column number in error messages.
"-qtrigraph" on page 249	None.	Enables the recognition of trigraph key combinations to represent characters not found on some keyboards.
"-U" on page 251	None.	Undefines a macro defined by the compiler or by the -D compiler option.
"-qutf" on page 256	None.	Enables recognition of UTF literal syntax.

Template control (C++ only)

You can use these options to control how the C++ compiler handles templates.

Table 12. C++ template options

Option name	Equivalent pragma name	Description
"-qtempinc (C++ only)" on page 239	None.	Generates separate template include files for template functions and class declarations, and places these files in a directory which can be optionally specified.
"-qtemplatedepth (C++ only)" on page 240	None.	Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.
"-qtemplaterecompile (C++ only)" on page 241	None.	Helps manage dependencies between compilation units that have been compiled using the -qtemplateregistry compiler option.

Table 12. C++ template options (continued)

Option name	Equivalent pragma name	Description
"-qtemplateregistry (C++ only)" on page 242	None.	Maintains records of all templates as they are encountered in the source and ensures that only one instantiation of each template is made.
"-qtempmax (C++ only)" on page 243	None.	Specifies the maximum number of template include files to be generated by the -qtempinc option for each header file.
"-qtmplinst (C++ only)" on page 247	None.	Manages the implicit instantiation of templates.
"-qtmplparse (C++ only)" on page 248	None.	Controls whether parsing and semantic checking are applied to template definitions.

Floating-point and integer control

Specifying the details of how your applications perform calculations can allow you to take better advantage of your system's floating-point performance and precision, including how to direct rounding. However, keep in mind that strictly adhering to IEEE floating-point specifications can impact the performance of your application. Using the options in the following table, you can control trade-offs between floating-point performance and adherence to IEEE standards.

Table 13. Floating-point and integer control options

Option name	Equivalent pragma name	Description
"-qbitfields" on page 64	None.	Specifies whether bit fields are signed or unsigned.
"-qchars" on page 69	#pragma options chars, #pragma chars	Determines whether all variables of type char are treated as either signed or unsigned.
"-qenum" on page 90	#pragma options enum, #pragma enum	Specifies the amount of storage occupied by enumerations.
"-qfloat" on page 98	#pragma options float	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.
"-qlonglit (PPU only)" on page 166	None.	In 64-bit mode, promotes literals with implicit type of int to long.
"-y" on page 266	None.	Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.

Object code control

These options affect the characteristics of the object code, preprocessed code, or other output generated by the compiler.

Table 14. Object code control options

Option name	Equivalent pragma name	Description
"-q32, -q64 (PPU only)" on page 49	None.	Selects either 32-bit or 64-bit compiler mode.
"-qalloca, -ma (C only)" on page 55	#pragma alloca	Provides an inline definition of system function <code>alloca</code> when it is called from source code that does not include the <code>alloca.h</code> header.
"-qcommon (PPU only)" on page 72	None.	Controls where uninitialized global variables are allocated.
"-qeh (C++ only) (PPU only)" on page 89	None.	Controls whether exception handling is enabled in the module being compiled.
"-qinlglue (PPU only)" on page 131	#pragma options inlglue	When used with -O2 or higher optimization, inlines glue code that optimizes external function calls in your application.
"-qkeepinlines (C++ only)" on page 143	None.	Keeps or discards definitions for unreferenced extern inline functions.
"-qpica" on page 190	None.	Generates Position-Independent Code suitable for use in shared libraries.
"-qpplina" on page 191	None.	When used in conjunction with the -E or -P options, enables or disables the generation of <code>#line</code> directives.
"-qpriorit (C++ only)" on page 194	#pragma options priority, #pragma priority	Specifies the priority level for the initialization of static objects.
"-qproto (C only)" on page 197	#pragma options proto	Specifies the linkage conventions for passing floating-point arguments to functions that have not been prototyped.
"-r" on page 200	None.	Produces a relocatable object, even though the file contains unresolved symbols.
"-qreserved_reg" on page 203	None.	Indicates that the given list of registers cannot be used during the compilation except as a stack pointer, frame pointer or in some other fixed role.
"-qro (PPU only)" on page 204	#pragma options ro, #pragma strings	Specifies the storage type for string literals.

Table 14. Object code control options (continued)

Option name	Equivalent pragma name	Description
"-qroconst (PPU only)" on page 206	#pragma options roconst	Specifies the storage location for constant values.
"-qrtti (C++ only) (PPU only)" on page 207	None.	Generates runtime type identification (RTTI) information for exception handling and for use by the typeid and dynamic_cast operators.
"-s" on page 208	None.	Strips the symbol table, line number information, and relocation information from the output file.
"-qsaveopt" on page 209	None.	Saves the command-line options used for compiling a source file, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.
"-qstatsym" on page 224	None.	Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the symbol table of the object file.
"-qtbtable (PPU only)" on page 238	#pragma options tbtable	Controls the amount of debugging traceback information that is included in the object files.
"-qthreaded (PPU only)" on page 244	None.	Indicates to the compiler whether it must generate threadsafe code.
"-qtls (PPU only)" on page 245	None.	Enables recognition of the __thread storage class specifier, which designates variables that are to be allocated threadlocal storage; and specifies the threadlocal storage model to be used.
"-qxcall" on page 264	None.	Generates code to treat static functions within a compilation unit as if they were external functions.

Error checking and debugging

The options in this category allow you to detect and correct problems in your source code. In some cases, these options can alter your object code, increase your

compile time, or introduce runtime checking that can slow down the execution of your application. The option descriptions indicate how extra checking can impact performance.

To control the amount and type of information you receive regarding the behavior and performance of your application, consult the options in “Listings, messages, and compiler information” on page 39.

For information on debugging optimized code, see the *XL C/C++ Optimization and Programming Guide*.

Table 15. Error checking and debugging options

Option name	Equivalent pragma name	Description
“-# (pound sign)” on page 48	None.	Previews the compilation steps specified on the command line, without actually invoking any compiler components.
“-qcheck” on page 70	#pragma options check	Generates code that performs certain types of runtime checking.
“-qfltrap (PPU only)” on page 103	#pragma options fltrap	Determines the types of floating-point exception conditions to be detected at run time
“-qformat” on page 105	None.	Warns of possible problems with string input and output format specifications.
“-qfullpath” on page 106	#pragma options fullpath	When used with the -g or -qlinedebug option, this option records the full, or absolute, path names of source and include files in object files compiled with debugging information, so that debugging tools can correctly locate the source files.
“-g” on page 108	None.	Generates debug information for use by a symbolic debugger.
“-qhalt” on page 113	#pragma options halt	Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity you specify.
“-qhaltonmsg (C++ only)” on page 114	None.	Stops compilation before producing any object, executable, or assembler source files if a specified error message is generated.
“-qinfo” on page 124	#pragma options info, #pragma info	Produces or suppresses groups of informational messages.

Table 15. Error checking and debugging options (continued)

Option name	Equivalent pragma name	Description
"-qinitauto" on page 130	#pragma options initauto	Initializes uninitialized automatic variables to a specific value, for debugging purposes.
"-qkeepparm" on page 144	None.	When used with -O2 or higher optimization, specifies whether function parameters are stored on the stack.
"-qlinedebug" on page 163	None.	Generates only line number and source file name information for a debugger.
"-qmaxerr" on page 170	None.	Halts compilation when a specified number of errors of a specified severity level or higher is reached.
"-qnewcheck (C++ only)" on page 177	None.	Specifies whether the compiler inserts a check to determine whether a null pointer has been returned by a call to a version of operator new or operator new[] with a non-empty throw specification.
"-qoptdebug" on page 182	None.	When used with high levels of optimization, produces files containing optimized pseudocode that can be read by a debugger.
"-qsymtab (C only)" on page 234	None.	Determines the information that appears in the symbol table.
"-qsyntaxonly (C only)" on page 235	None.	Performs syntax checking without generating an object file.
"-qwarn64 (PPU only)" on page 263	None.	Enables checking for possible data conversion problems between 32-bit and 64-bit compiler modes.

Listings, messages, and compiler information

The options in this category allow you control over the listing file, as well as how and when to display compiler messages. You can use these options in conjunction with those described in "Error checking and debugging" on page 37 to provide a more robust overview of your application when checking for errors and unexpected behavior.

Table 16. Listings and messages options

Option name	Equivalent pragma name	Description
"-qattr" on page 62	#pragma options attr	Produces a compiler listing that includes the attribute component of the attribute and cross-reference section of the listing.
"-qdump_class_hierarchy (C++ only)" on page 85	None.	Dumps a representation of the hierarchy and virtual function table layout of each class object to a file.
"-qflag" on page 97	#pragma options flag, C++ "#pragma report (C++ only)" on page 304	Limits the diagnostic messages to those of a specified severity level or higher.
"-qlist" on page 164	#pragma options list	Produces a compiler listing file that includes an object or assembly listing.
"-qlistopt" on page 165	None.	Produces a compiler listing file that includes all options in effect at the time of compiler invocation.
"-qphsinfo" on page 189	None.	Reports the time taken in each compilation phase to standard output.
"-qprint" on page 193	None.	Enables or suppresses listings.
"-qreport" on page 202	None.	Produces listing files that show how sections of code have been optimized.
"-qshowinc" on page 211	#pragma options showinc	When used with -qsource option to generate a listing file, selectively shows user or system header files in the source section of the listing file.
"-qsource" on page 218	#pragma options source	Produces a compiler listing file that includes the source section of the listing and provides additional source information when printing error messages.
"-qsrcmsg (C only)" on page 221	None.	Adds the corresponding source code lines to diagnostic messages generated by the compiler.
"-qsuppress" on page 231	None.	Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.

Table 16. Listings and messages options (continued)

Option name	Equivalent pragma name	Description
"-v, -V" on page 257	None.	Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.
"-qversion" on page 258	None.	Displays the version and release of the compiler being invoked.
"-w" on page 261	None.	Suppresses informational, language-level and warning messages.
"-qxref" on page 265	#pragma options xref	Produces a compiler listing that includes the cross-reference component of the attribute and cross-reference section of the listing.

Optimization and tuning

The options in this category allow you to control the optimization and tuning process, which can improve the performance of your application at run time.

Remember that not all options benefit all applications. Trade-offs sometimes occur between an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide.

You can also control some of these options, such as **Optimize**, **-qcompact**, or **-qstrict**, with an **option_override** pragma.

In addition to the option descriptions in this section, consult the *XL C/C++ Optimization and Programming Guide* for a details on the optimization and tuning process as well as writing optimization-friendly source code.

Table 17. Optimization and tuning options

Option name	Equivalent pragma name	Description
"-qaggrcopy" on page 50	None.	Enables destructive copy operations for structures and unions.
"-qalias" on page 51	None.	Indicates whether a program contains certain categories of aliasing or does not conform to C/C++ standard aliasing rules. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage location..
"-qarch" on page 57	None.	Specifies the processor architecture for which the code (instructions) should be generated.

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qatomic_updates (SPU only)" on page 61	None.	Enables atomic updates to the software cache manager used in effective address space support.
"-qcache (PPU only)" on page 67	None.	When specified with -O4 , -O5 , or -qipa , specifies the cache configuration for a specific execution machine.
"-qcompact" on page 74	#pragma options compact	Avoids optimizations that increase code size.
"-qdataimported, -qdatalocal, -qtocdata (PPU only)" on page 81	None.	Marks data as local or imported in 64-bit compilations.
"-qdirectstorage (PPU only)" on page 84	None.	Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.
"-qenablevmx (PPU only)" on page 94	None.	Enables generation of vector instructions.
"-qfdpr" on page 96	None.	Provides object files with information that the IBM Feedback Directed Program Restructuring (FDPR) performance-tuning utility needs to optimize the resulting executable file.
"-qhot" on page 115	#pragma nosimd, #pragma novector	Performs high-order loop analysis and transformations (HOT) during optimization.
"-qignerrno" on page 120	#pragma options ignerrno	Allows the compiler to perform optimizations that assume errno is not modified by system calls.
"-qipa" on page 132	None.	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).
"-qisolated_call" on page 141	#pragma options isolated_call, #pragma isolated_call	Specifies functions in the source file that have no side effects other than those implied by their parameters.
"-qlargepage" on page 160	None.	Takes advantage of large pages for applications designed to execute in a large page memory environment.
"-qlibansi" on page 162	#pragma options libansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qmaxmem" on page 172	#pragma options maxmem	Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.
"-qminimaltoc (PPU only)" on page 175	None.	Controls the generation of the table of contents (TOC), which the compiler creates for an executable file in 64-bit compilation mode.
"-O, -qoptimize" on page 179	#pragma options optimize	Specifies whether to optimize code during compilation and, if so, at which level.
"-p, -pg, -qprofile (PPU only)" on page 183	None.	Prepares the object files produced by the compiler for profiling.
"-qpdf1, -qpdf2 (PPU only)" on page 187	None.	Tunes optimizations through <i>profile-directed feedback</i> (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.
"-qprefetch (PPU only)" on page 192	None.	Inserts prefetch instructions automatically where there are opportunities to improve code performance.
"-qprocimported, -qproclocal, -qprocunknown (PPU only)" on page 195	#pragma options procimported, #pragma options proclocal, #pragma options procunknown	Marks functions as local, imported, or unknown in 64-bit compilations.
"-Q, -qinline " on page 198	None.	Attempts to inline functions instead of generating calls to those functions, for improved performance.
"-qsmallstack" on page 213	None.	Reduces the size of the stack frame.
"-qstrict" on page 227	#pragma options strict	Ensures that optimizations done by default at optimization levels -O3 and higher, and, optionally at -O2 , do not alter the semantics of a program.

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qstrict_induction" on page 231	None.	Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.
"-qswcache_size (SPU only)" on page 233	None.	Sets the size of the software cache used in effective address space support.
"-qtune" on page 250	#pragma options tune	Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.
"-qunroll" on page 252	#pragma options unroll, #pragma unroll	Controls loop unrolling, for improved performance.
"-qunwind" on page 255	None.	Specifies whether the call stack can be unwound by code looking through the saved registers on the stack.

Linking

Though linking occurs automatically, the options in this category allow you to direct input and output to the linker, controlling how the linker processes your object files.

Table 18. Linking options

Option name	Equivalent pragma name	Description
"-qcr" on page 77	None.	Specifies whether system startup files are to be linked.
"-e" on page 86	None.	When used together with the -qmkshrobj , specifies an entry point for a shared object.
"-L" on page 147	None.	At link time, searches the directory path for library files specified by the -l option.
"-l" on page 146	None.	Searches for the specified library file, <i>libkey.so</i> , and then <i>libkey.a</i> for dynamic linking, or just for <i>libkey.a</i> for static linking.

Table 18. Linking options (continued)

Option name	Equivalent pragma name	Description
"-qlib" on page 161	None.	Specifies whether standard system libraries and XL C/C++ libraries are to be linked.
"-R" on page 201	None.	At link time, writes search paths for shared libraries into the executable, so that these directories are searched at program run time for any required shared libraries.
"-qstaticlink" on page 223	None.	Controls how shared and non-shared runtime libraries are linked into an application.
"-qstdmain (SPU only)" on page 226	None.	Links your SPU program with the startup routines required to use C99-style main function arguments.

Portability and migration

The options in this category can help you maintain application behavior compatibility on past, current, and future hardware, operating systems and compilers, or help move your applications to an XL compiler with minimal change.

Table 19. Portability and migration options

Option name	Equivalent pragma name	Description
"-qabi_version (C++ only)" on page 49	None.	Specifies the version of the C++ application binary interface (ABI) version used during compilation. This option is provided for compatibility with different levels of GNU C++.
"-qalign" on page 53	#pragma options align, #pragma align	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.
"-qgenproto (C only)" on page 111	None.	Produces prototype declarations from K&R function definitions or function definitions with empty parentheses, and displays them to standard output.
"-qupconv (C only)" on page 255	#pragma options upconv	Specifies whether the unsigned specification is preserved when integral promotions are performed.

Compiler customization

The options in this category allow you to specify alternate locations for compiler components, configuration files, standard include directories, and internal compiler operation. You should only need to use these options in specialized installation or testing scenarios.

Table 20. Compiler customization options

Option name	Equivalent pragma name	Description
"-qasm_as (PPU only)" on page 60	None.	Specifies the path and flags used to invoke the assembler in order to handle assembler code in an asm assembly statement.
"-B" on page 63	None.	Determines substitute path names for XL C/C++ executables such as the compiler, assembler, linker, and preprocessor.
"-qcomplexgccincl" on page 74	#pragma complexgcc	Specifies whether to use GCC parameter-passing conventions for complex data types (equivalent to enabling -qfloat=complexgcc) for selected include files only.
"-qc_stdinc (C only)" on page 78	None.	Changes the standard search location for the XL C header files.
"-qcpp_stdinc (C++ only)" on page 79	None.	Changes the standard search location for the XL C++ header files.
"-F" on page 94	None.	Names an alternative configuration file or stanza for the compiler.
"-qgcc_c_stdinc (C only)" on page 109	None.	Changes the standard search location for the GNU C system header files.
"-qgcc_cpp_stdinc (C++ only)" on page 110	None.	Changes the standard search location for the GNU C++ system header files.
"-qpath" on page 185	None.	Determines substitute path names for XL C/C++ executables such as the compiler, assembler, and linker.
"-qspill" on page 220	#pragma options spill	Specifies the size (in bytes) of the register spill space, the internal program storage areas used by the optimizer for register spills to storage.
"-t" on page 236	None.	Applies the prefix specified by the -B option to the designated components.
"-W" on page 262	None.	Passes the listed options to a component that is executed during compilation.

Individual option descriptions

This section contains descriptions of the individual compiler options available in XL C/C++.

For each option, the following information is provided:

Category

The functional category to which the option belongs is listed here.

Pragma equivalent

Many compiler options allow you to use an equivalent pragma directive to apply the option's functionality within the source code, limiting the scope of the option's application to a single source file, or even selected sections

of code. Where an option supports the **#pragma options** *option_name* and/or **#pragma name** form of the directive, this is indicated.

Purpose

This section provides a brief description of the effect of the option (and equivalent pragmas), and why you might want to use it.

Syntax

This section provides the syntax for the option, and where an equivalent **#pragma name** is supported, the specific syntax for the pragma. Syntax for **#pragma options** *option_name* forms of the pragma is not provided, as this is normally identical to that of the option. Note that you can also use the C99-style `_Pragma` operator form of any pragma; although this syntax is not provided in the option descriptions. For complete details on pragma syntax, see “Pragma directive syntax” on page 269

Defaults

In most cases, the default option setting is clearly indicated in the syntax diagram. However, for many options, there are multiple default settings, depending on other compiler options in effect. This section indicates the different defaults that may apply.

Parameters

This section describes the suboptions that are available for the option and pragma equivalents, where applicable. For suboptions that are specific to the command-line option or to the pragma directive, this is indicated in the descriptions.

Usage This section describes any rules or usage considerations you should be aware of when using the option. These can include restrictions on the option’s applicability, valid placement of pragma directives, precedence rules for multiple option specifications, and so on.

Predefined macros

Many compiler options set macros that are protected (that is, cannot be undefined or redefined by the user). Where applicable, any macros that are predefined by the option, and the values to which they are defined, are listed in this section. A reference list of these macros (as well as others that are defined independently of option setting) is provided in Chapter 5, “Compiler predefined macros,” on page 325

Examples

Where appropriate, examples of the command-line syntax and pragma directive use are provided in this section.

-+ (plus sign) (C++ only)

Category

Input control

Pragma equivalent

None.

Purpose

Compiles any file as a C++ language file.

This option is equivalent to the **-qsource-type=c++** option.

Syntax

►► -+ ◀◀

Usage

You can use `++` to compile a file with any suffix other than `.a`, `.o`, `.so`, `.S` or `.s`. If you do not use the `++` option, files must have a suffix of `.C` (uppercase C), `.cc`, `.cp`, `.cpp`, `.cxx`, or `.c++` to be compiled as a C++ file. If you compile files with suffix `.c` (lowercase c) without specifying `++`, the files are compiled as a C language file.

The `++` option should not be used together with the `-qsourcetype` option.

Predefined macros

None.

Examples

To compile the file `myprogram.cplsp1s` as a C++ source file, enter:

```
ppuxlc++ ++ myprogram.cplsp1s
```

Related information

- “`-qsourcetype`” on page 219

-# (pound sign)

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Previews the compilation steps specified on the command line, without actually invoking any compiler components.

When this option is enabled, information is written to standard output, showing the names of the programs within the preprocessor, compiler, and linker that would be invoked, and the default options that would be specified for each program. The preprocessor, compiler, and linker are not invoked.

Syntax

►► -# ◀◀

Usage

You can use this command to determine the commands and files that will be involved in a particular compilation. It avoids the overhead of compiling the source code and overwriting any existing files, such as `.lst` files.

This option displays the same information as **-v**, but does not invoke the compiler. The **-#** option overrides the **-v** option.

Predefined macros

None.

Examples

To preview the steps for the compilation of the source file `myprogram.c`, enter:

```
invocation myprogram.c -#
```

Related information

- “-v, -V” on page 257

-q32, -q64 (PPU only)

Category

Object code control

Pragma equivalent

None.

Purpose

Selects either 32-bit or 64-bit compiler mode.

Note: You can only compile 64-bit code using one of the **ppu** invocation commands, such as **ppuxlc**. A program that targets both PPU and SPU and that is compiled using one of the **cbe** invocation commands, such as **cbexlc** must be 32-bit.

Syntax

►►  **-q**  ►►

Defaults

-q32

Predefined macros

`__64BIT__` is defined to 1 when **-q64** is in effect; otherwise, it is undefined.

-qabi_version (C++ only)

Category

Portability and migration

Pragma equivalent

None.

Purpose

Specifies the version of the C++ application binary interface (ABI) version used during compilation. This option is provided for compatibility with different levels of GNU C++.

Syntax

►► -qabi_version=1 2►►

Defaults

-qabi_version=2

Parameters

- 1 Specifies the same C++ ABI behavior as in GNU C++ 3.2.
- 2 Specifies the same C++ ABI behavior as in GNU C++ 3.4.

Predefined macros

None.

-qaggrcopy

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables destructive copy operations for structures and unions.

Syntax

►► -qaggrcopy=nooverlap overlap►►

Defaults

-qaggrcopy=nooverlap

Parameters

overlap | **nooverlap**

nooverlap assumes that the source and destination for structure and union assignments do not overlap, allowing the compiler to generate faster code.
overlap inhibits these optimizations.

Predefined macros

None.

-qalias

Category

Optimization and tuning

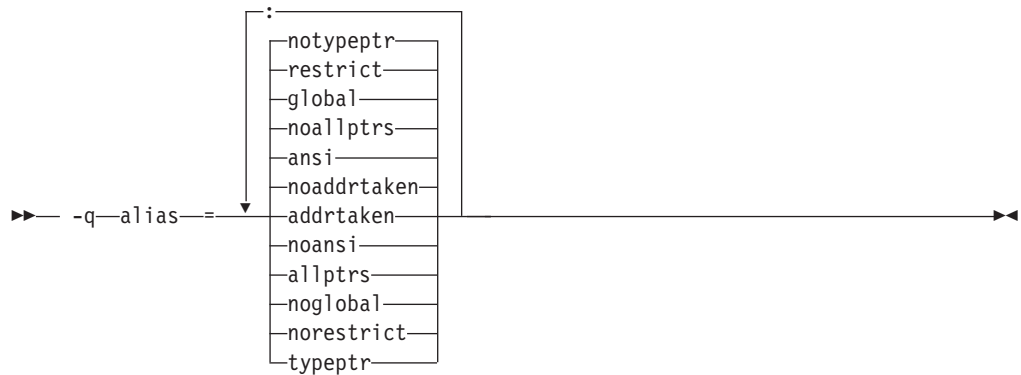
Pragma equivalent

None

Purpose

Indicates whether a program contains certain categories of aliasing or does not conform to C/C++ standard aliasing rules. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage location.

Syntax



Defaults

- **C++** `-qalias=noaddrtaken:noallptrs:ansi:global:restrict:notypeptr`
- **C** `-qalias=noaddrtaken:noallptrs:ansi:global:restrict:notypeptr` for all invocation commands except `cc`.
`-qalias=noaddrtaken:noallptrs:noansi:global:restrict:notypeptr` for the `cc` invocation command.

Parameters

`addrtaken` | `noaddrtaken`

When **`addrtaken`** is in effect, variables are disjoint from pointers unless their address is taken. Any class of variable for which an address has *not* been recorded in the compilation unit will be considered disjoint from indirect access through pointers.

When **`noaddrtaken`** is specified, the compiler generates aliasing based on the aliasing rules that are in effect.

`allptrs` | `noallptrs`

When **`allptrs`** is in effect, pointers are never aliased (this also implies

-qalias=typeptr). Specifying **allptrs** is an assertion to the compiler that no two pointers point to the same storage location. These suboptions are only valid if **ansi** is also specified.

ansi | **noansi**

When **ansi** is in effect, type-based aliasing is used during optimization, which restricts the lvalues that can be safely used to access a data object. The optimizer assumes that pointers can *only* point to an object of the same type. This suboption has no effect unless you also specify an optimization option.

When **noansi** is in effect, the optimizer makes worst case aliasing assumptions. It assumes that a pointer of a given type can point to an external object or any object whose address is already taken, regardless of type.

global | **noglobal**

When **global** is in effect, type-based aliasing rules are enabled during IPA link-time optimization across compilation units. Both **-qipa** and **-qalias=ansi** must be enabled for **-qalias=global** to have an effect. Specifying **noglobal** disables type-based aliasing rules.

-qalias=global produces better performance at higher optimization levels and also better link-time performance. If you use **-qalias=global**, it is recommended that you compile as much as possible of the application with the same version of the compiler to maximize the effect of the suboption on performance.

restrict | **norestrict**

When **restrict** is in effect, optimizations for pointers qualified with the **restrict** keyword are enabled. Specifying **norestrict** disables optimizations for **restrict**-qualified pointers.

-qalias=restrict is independent from other **-qalias** suboptions. Using the **-qalias=restrict** option will usually result in performance improvements for code that uses **restrict**-qualified pointers. Note, however, that using **-qalias=restrict** requires that restricted pointers be used correctly; if they are not, compile-time and runtime failures may result. You can use **norestrict** to preserve compatibility with code compiled with versions of the compiler previous to V9.0.

typeptr | **notypeptr**

When **typeptr** is in effect, pointers to different types are never aliased. Specifying **typeptr** is an assertion to the compiler that no two pointers of different types point to the same storage location. These suboptions are only valid if **ansi** is also specified.

Usage

-qalias makes assertions to the compiler about the code that is being compiled. If the assertions about the code are false, then the code generated by the compiler may result in unpredictable behaviour when the application is run.

The following are not subject to type-based aliasing:

- Signed and unsigned types. For example, a pointer to a signed `int` can point to an unsigned `int`.
- Character pointer types can point to any type.
- Types qualified as `volatile` or `const`. For example, a pointer to a `const int` can point to an `int`.

Predefined macros

None.

Examples

To specify worst-case aliasing assumptions when compiling `myprogram.c`, enter:

```
invocation myprogram.c -O -qalias=noansi
```

Related information

- “-qipa” on page 132
- “#pragma disjoint” on page 280
- *Type-based aliasing* in the *XL C/C++ Language Reference*
- *The restrict type qualifier* in the *XL C/C++ Language Reference*

-qalign

Category

Portability and migration

Pragma equivalent

#pragma options align, #pragma align

Purpose

Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.

Syntax

Option syntax

►► -q-align=linuxppc
bit_packed►►

Pragma syntax

►► #pragma align-(linuxppc
bit_packed
reset)►►

Defaults

linuxppc

Parameters

bit_packed

Bit field data is packed on a bitwise basis without respect to byte boundaries. **-qalign=bit_packed** may cause problems on the SPU if the bitfield container is spread across a natural boundary (i.e. 17-bit bitfield crossing a 4-byte boundary).

linuxppc

Uses GNU C/C++ alignment rules to maintain binary compatibility with GNU C/C++ objects.

reset (pragma only)

Discards the current pragma setting and reverts to the setting specified by the previous pragma directive. If no previous pragma was specified, reverts to the command-line or default option setting.

Usage

If you use the **-qalign** option more than once on the command line, the last alignment rule specified applies to the file.

The pragma directives override the **-qalign** compiler option setting for a specified section of program source code. The pragmas affect all aggregate definitions that appear after a given pragma directive; if a pragma is placed inside a nested aggregate, it applies only to the definitions that follow it, not to any containing definitions. Any aggregate variables that are declared use the alignment rule that applied at the point at which the aggregate was *defined*, regardless of pragmas that precede the declaration of the variables. See below for examples.

For a complete explanation of the option and pragma parameters, as well as usage considerations, see "Aligning data" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Examples

The following examples show the interaction of the option and pragmas. Assuming compilation with the command `xlc file2.c`, the following example shows how the pragma affects only an aggregate *definition*, not subsequent declarations of variables of that aggregate type.

```
/* file2.c The default alignment rule is in effect */

typedef struct A A2;

#pragma options align=bit_packed /* bit_packed alignment rules are now in effect */
struct A {
    int a;
    char c;
}; #pragma options align=reset /* Default alignment rules are in effect again */

struct A A1; /* A1 and A3 are aligned using bit_packed alignment rules since */
A2 A3;      /* this rule applied when struct A was defined */
```

Assuming compilation with the command `xlc file.c -qalign=bit_packed`, the following example shows how a pragma embedded in a nested aggregate definition affects only the definitions that follow it.

```
/* file2.c The default alignment rule in effect is bit_packed */

struct A {
    int a;
    #pragma options align=linuxppc /* Applies to B; A is unaffected */
    struct B {
        char c;
    };
};
```

```

double d;
    } BB; /* BB uses linuxppc alignment rules */
} AA; /* AA uses bit_packed alignment rules */

```

Related information

- “#pragma pack” on page 298
- “Aligning data” in the *XL C/C++ Optimization and Programming Guide*
- The __align specifier “The __align specifier” in the *XL C/C++ Language Reference*
- “The aligned variable attribute” in the *XL C/C++ Language Reference*
- “The packed variable attribute” in the *XL C/C++ Language Reference*

-qalloca, -ma (C only)

Category

Object code control

Pragma equivalent

#pragma alloca

Purpose

Provides an inline definition of system function `alloca` when it is called from source code that does not include the `alloca.h` header.

The function `void* alloca(size_t size)` dynamically allocates memory, similarly to the standard library function `malloc`. The compiler automatically substitutes calls to the system `alloca` function with an inline built-in function `__alloca` in any of the following cases:

- You include the header file `alloca.h`
- You compile with **-Dalloca=__alloca**
- You directly call the built-in function using the form `__alloca`

The **-qalloca** and **-ma** options and **#pragma alloca** directive provide the same functionality in C only, if any of the above methods are not used.

Syntax

Option syntax

```

▶▶ [ -qalloca | -ma ]

```

Pragma syntax

```

▶▶ #pragma alloca

```

Defaults

Not applicable.

Usage

If you do not use any of the above-mentioned methods to ensure that calls to `alloca` are replaced with `__alloca`, `alloca` is treated as a user-defined identifier rather than as a built-in function.

Once specified, **#pragma alloca** applies to the rest of the file and cannot be disabled. If a source file contains any functions that you want compiled without **#pragma alloca**, place these functions in a different file.

You may want to consider using a C99 variable length array in place of `alloca`.

Predefined macros

None.

Examples

To compile `myprogram.c` so that calls to the function `alloca` are treated as inline, enter:

```
invocation myprogram.c -qalloca
```

Related information

- “-D” on page 80
- “__alignx” on page 362

-qaltivec

Category

Language element control

Pragma equivalent

None.

Purpose

Enables compiler support for vector data types and operators.

See the *XL C/C++ Language Reference* for complete documentation of vector data types.

Syntax

```
►► — -q ————— ◄◄  
      altivec  
      noaltivec
```

Defaults

-qaltivec

Predefined macros

`__ALTIVEC__` is defined to 1 and `__VEC__` is defined to 10205 when `-qaltivec` is in effect; otherwise, they are undefined.

Related information

- *AltiVec Technology Programming Interface Manual*, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

-qarch

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Specifies the processor architecture for which the code (instructions) should be generated.

Syntax



Defaults

- `-qarch=cellppu` for compilation targeting the PPU when compiled using one of the ppu invocation commands, such as `ppuxlc`.
- `-qarch=cellspu` for compilation targeting the SPU when compiled using one of the spu invocation commands, such as `spuxlc`.
- `-qarch=cell` for compilation targeting either the PPU or SPU when compiled using one of the cbe invocation commands, such as `cbexlc`.

Parameters

auto

Automatically detects the specific architecture of the compiling machine. It assumes that the execution environment will be the same as the compilation environment. This option is implied if the `-O4` or `-O5` option is set or implied.

cell

Produces object code containing instructions that can be run on either the PPU or SPU when compiled using one of the cbe invocation commands, such as `cbexlc`.

celledp

Produces object code containing instructions that can be run on either the PPU or SPU with enhanced double precision when compiled using one of the cbe invocation commands, such as `cbexlc`.

cellppu (PPU only)

Produces object code containing instructions that will run on the PPU when compiled using one of the ppu invocation commands, such as **ppuxlc**.

cellspu (SPU only)

Produces object code containing instructions that will run on the SPU when compiled using one of the spu invocation commands, such as **spuxlc**.

edp (SPU only)

Produces object code containing instructions that will run on the enhanced CBEA-compliant processor with a double precision SPU when compiled using one of the spu invocation commands, such as **spuxlc**.

Predefined macros

See “Macros related to architecture settings” on page 330 for a list of macros that are predefined by **-qarch** suboptions.

Related information

- “-qtune” on page 250
- “-q32, -q64 (PPU only)” on page 49
- “Optimizing your applications” in the *XL C/C++ Optimization and Programming Guide*

-qasm**Category**

Language element control

Pragma equivalent

None.

Purpose

Controls the interpretation of and subsequent generation of code for assembler language extensions.

When **-qasm** is in effect, the compiler generates code for assembly statements in the source code. Suboptions specify the syntax used to interpret the content of the assembly statement.

Note: The system assembler program must be available for this command to have effect.

Syntax**-qasm syntax — C**

-qasm syntax — C++



Defaults

-qasm=gcc

Parameters

gcc

Instructs the compiler to recognize the extended GCC syntax and semantics for assembly statements.

C++

stdcpp

Reserved for possible future use.

Specifying **-qasm** without a suboption is equivalent to specifying the default.

Usage

C

The token `asm` is not a C language keyword. Therefore, at language levels **stdc89** and **stdc99**, which enforce strict compliance to the C89 and C99 standards, respectively, the option **-qkeyword=asm** must also be specified to compile source that generates assembly code. At all other language levels, the token `asm` is treated as a keyword unless the option **-qnokeyword=asm** is in effect. In C, the compiler-specific variants `__asm` and `__asm__` are keywords at all language levels and cannot be disabled.

C++

The tokens `asm`, `__asm`, and `__asm__` are keywords at all language levels. Suboptions of **-qnokeyword=token** can be used to disable each of these reserved words individually.

For detailed information on the syntax and semantics of inline `asm` statements, see "Inline assembly statements" in the *XL C/C++ Language Reference*.

Predefined macros

- **C** `__IBM_GCC_ASM` is predefined to 1 when `asm` is recognized as a keyword and assembler code is generated; that is, at all language levels except **stdc89** | **stdc99**, or when **-qkeyword=asm** is in effect, and when **-qasm[=gcc]** is in effect. It is predefined to 0 when `asm` is recognized as a keyword but assembler code is not generated; that is, at all language levels except **stdc89** | **stdc99**, or when **-qkeyword=asm** is in effect, and when **-qnoasm** is in effect. It is undefined when the **stdc89** | **stdc99** language level or **-qnokeyword=asm** is in effect.
- **C++** `__IBM_GCC_ASM` is predefined to 1 when `asm` is recognized as a keyword and assembler code is generated; that is, at all language levels, and when **-qasm[=gcc]** is in effect. It is predefined to 0 when `asm` is recognized as a keyword but assembler code is not generated; that is, at all language levels, and

when **-qnoasm[=gcc]** is in effect. It is undefined when **-qnoasm=stdcpp** is in effect. `__IBM_STDCPP_ASM` is predefined to 0 when **-qnoasm=stdcpp** is in effect; otherwise it is undefined.

Examples

The following code snippet shows an example of the GCC conventions for `asm` syntax in inline statements:

```
int a, b, c;
int main() {
    asm("add %0, %1, %2" : "=r"(a) : "r"(b), "r"(c) );
}
```

Related information

- “-qasm_as (PPU only)”
- “-qlanglvl” on page 148
- “-qkeyword” on page 145
- “Inline assembly statements” in the *XL C/C++ Language Reference*
- “Keywords for language extensions”

-qasm_as (PPU only)

Category

Compiler customization

Pragma equivalent

None.

Purpose

Specifies the path and flags used to invoke the assembler in order to handle assembler code in an `asm` assembly statement.

Normally the compiler reads the location of the assembler from the configuration file; you can use this option to specify an alternate assembler program and flags to pass to that assembler.

Syntax

```
►► -qasm_as= [path] [flags] ►►
```

The diagram shows the syntax of the `-qasm_as` option. It consists of the option name followed by an equals sign, then a bracketed section containing a `path` and a `flags` section, all enclosed in a larger bracket.

Defaults

By default, the compiler invokes the assembler program defined for the `as` command in the compiler configuration file.

Parameters

path

The full path name of the assembler to be used.

flags

A space-separated list of options to be passed to the assembler for assembly statements. Quotation marks must be used if spaces are present.

Predefined macros

None.

Examples

To instruct the compiler to use the assembler program at `/bin/as` when it encounters inline assembler code in `myprogram.c`, enter:

```
invocation myprogram.c -qasm_as=/bin/as
```

To instruct the compiler to pass some additional options to the assembler at `/bin/as` for processing inline assembler code in `myprogram.c`, enter:

```
invocation myprogram.c -qasm_as="/bin/as -a64 -l a.lst"
```

Related information

- “-qasm” on page 58

-qatomic_updates (SPU only)

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables atomic updates to the software cache manager used in effective address space support.

Atomic updates to the software cache ensure that the updates from SPU to PPU are done with no interleaving from other processors, so concurrent changes to adjacent bytes in the cache line are not lost. Atomic updates are safer, but also slower than non-atomic updates.

Syntax

►► — -q — atomic_updates —————►
 └─noatomic_updates─┘

Defaults

-qatomic_updates is the default setting when compiling an SPU program using one of the `spu` invocation commands, such as **spuxlc**.

-qnoatomic_updates is the default when compiling a program that targets both PPU and SPU using one of the `cbe` invocation commands, such as **cbxlc**.

Parameters

full

Reports all identifiers in the program. If you specify **attr** without this suboption, only those identifiers that are used are reported.

Usage

If **-qattr** is specified after **-qattr=full**, it has no effect; the full listing is produced.

This option does not produce a cross-reference listing unless you also specify **-qxref**.

The **-qnoprint** option overrides this option.

Predefined macros

None.

Examples

To compile the program `myprogram.c` and produce a compiler listing of all identifiers, enter:

```
invocation myprogram.c -qxref -qattr=full
```

Related information

- “-qprint” on page 193
- “-qxref” on page 265

-B

Category

Compiler customization

Pragma equivalent

None.

Purpose

Determines substitute path names for XL C/C++ executables such as the compiler, assembler, linker, and preprocessor.

You can use this option if you want to keep multiple levels of some or all of the XL C/C++ executables and have the option of specifying which one you want to use. However, it is recommended that you use the **-qpath** option to accomplish this instead.

Syntax

►► -B prefix ◀◀

Defaults

The default paths for the compiler executables are defined in the compiler configuration file.

Parameters

prefix

Defines part of a path name for programs you can name with the **-t** option. You must add a slash (/). If you specify the **-B** option without the *prefix*, the default prefix is `/lib/o`.

Usage

The **-t** option specifies the programs to which the **-B** prefix name is to be appended; see “**-t**” on page 236 for a list of these. If you use the **-B** option without **-tprograms**, the prefix you specify applies to all of the compiler executables.

The **-B** and **-t** options override the **-F** option.

Predefined macros

None.

Examples

In this example, an earlier level of the compiler components is installed in the default installation directory. To test the upgraded product before making it available to everyone, the system administrator restores the latest installation image under the directory `/home/jim` and then tries it out with commands similar to:

```
invocation -tcbI -B/home/jim/opt/ibmcmp/xlc/cbe/10.1/bin/ test_suite.c
```

Once the upgrade meets the acceptance criteria, the system administrator installs it in the default installation directory.

Related information

- “**-qpath**” on page 185
- “**-t**” on page 236
- “Invoking the compiler” on page 1

-qbitfields

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

Specifies whether bit fields are signed or unsigned.

Syntax

►► — `-q-bitfields=` signed
unsigned —►►

Defaults

`-qbitfields=signed`

Parameters

signed

Bit fields are signed.

unsigned

Bit fields are unsigned.

Predefined macros

None.

-C

Category

Output control

Pragma equivalent

None.

Purpose

Prevents the completed object from being sent to the linker. With this option, the output is a `.o` file for each source file.

Syntax

►► — `-C` —►►

Defaults

By default, the compiler invokes the linker to link object files into a final executable.

Usage

When this option is in effect, the compiler creates an output object file, *file_name.o*, for each valid source file, such as *file_name.c*, *file_name.i*, *file_name.C*, *file_name.cpp*. You can use the `-o` option to provide an explicit name for the object file.

The `-c` option is overridden if the `-E`, `-P`, or `-qsyntaxonly` options are specified.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce an object file `myprogram.o`, but no executable file, enter the command:

```
invocation myprogram.c -c
```

To compile `myprogram.c` to produce the object file `new.o` and no executable file, enter:

```
invocation myprogram.c -c -o new.o
```

Related information

- “-E” on page 87
- “-o” on page 178
- “-P” on page 184
- “-qsyntaxonly (C only)” on page 235

-C, -C!

Category

Output control

Pragma equivalent

None.

Purpose

When used in conjunction with the **-E** or **-P** options, preserves or removes comments in preprocessed output.

When **-C** is in effect, comments are preserved. When **-C!** is in effect, comments are removed.

Syntax

►►

-C
-C!

 ◄◄

Defaults

-C

Usage

The **-C** option has no effect without either the **-E** or the **-P** option. If **-E** is specified, continuation sequences are preserved in the output. If **-P** is specified, continuation sequences are stripped from the output, forming concatenated output lines.

You can use the **-C!** option to override the **-C** option specified in a default makefile or configuration file.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce a file `myprogram.i` that contains the preprocessed program text including comments, enter:

```
invocation myprogram.c -P -C
```

Related information

- “-E” on page 87
- “-P” on page 184

-qcache (PPU only)

Category

Optimization and tuning

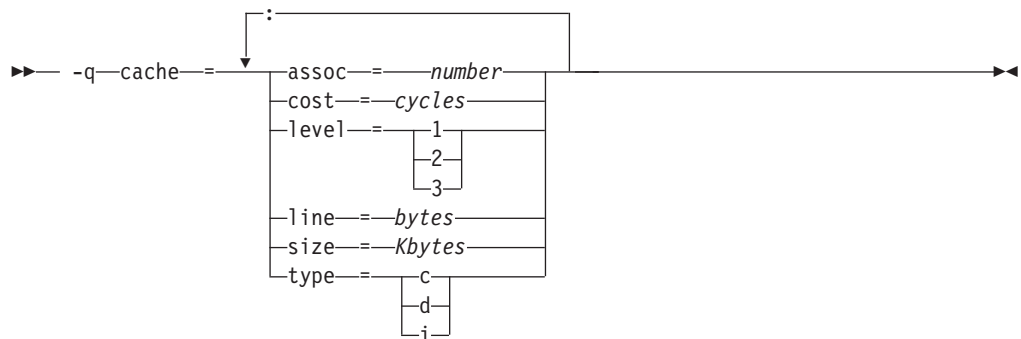
Pragma equivalent

None.

Purpose

When specified with **-O4**, **-O5**, or **-qipa**, specifies the cache configuration for a specific execution machine.

Syntax



Defaults

Automatically determined by the setting of the **-qtune** option.

```
-qcache=level=1:type=i:size=32:line=128:assoc=2:cost=4
```

```
-qcache=level=1:type=d:size=32:line=128:assoc=4:cost=4
```

```
-qcache=level=2:type=c:size=512:line=128:assoc=8:cost=36
```

Parameters

assoc

Specifies the set associativity of the cache.

number

Is one of:

- 0** Direct-mapped cache
- 1** Fully associative cache
- N>1** n-way set associative cache

cost

Specifies the performance penalty resulting from a cache miss.

cycles

level

Specifies the level of cache affected.

level

Is one of:

- 1** Basic cache
- 2** Level-2 cache
- 3** TLB

line

Specifies the line size of the cache.

bytes

An integer representing the number of bytes of the cache line.

size

Specifies the total size of the cache.

Kbytes

An integer representing the number of kilobytes of the total cache.

type

Specifies that the settings apply to the specified *cache_type*.

cache_type

Is one of:

- c** Combined data and instruction cache
- d** Data cache
- i** Instruction cache

Predefined macros

None.

Related information

- “-qcache (PPU only)” on page 67
- “-O, -qoptimize” on page 179
- “-qtune” on page 250
- “-qipa” on page 132

-qchars

Category

Floating-point and integer control

Pragma equivalent

#pragma options chars, #pragma chars

Purpose

Determines whether all variables of type char are treated as either signed or unsigned.

Syntax

Option syntax

►► -qchars=unsigned
signed►►

Pragma syntax

►► #pragma chars (unsigned
signed)►►

Defaults

-qchars=unsigned

Parameters

unsigned

Variables of type char are treated as unsigned char.

signed

Variables of type char are treated as signed char.

Usage

Regardless of the setting of this option or pragma, the type of char is still considered to be distinct from the types unsigned char and signed char for purposes of type-compatibility checking or C++ overloading.

The pragma must appear before any source statements. If the pragma is specified more than once in the source file, the first one will take precedence. Once specified, the pragma applies to the entire file and cannot be disabled; if a source file contains any functions that you want to compile without **#pragma chars**, place these functions in a different file.

Predefined macros

- `__CHAR_SIGNED` and `__CHAR_SIGNED__` are defined to 1 when **signed** is in effect; otherwise, it is undefined.

- `_CHAR_UNSIGNED` and `_CHAR_UNSIGNED__` are defined to 1 when **unsigned** is in effect; otherwise, they are undefined.

Examples

To treat all char types as signed when compiling `myprogram.c`, enter:

```
xlc myprogram.c -qchars=signed
```

-qcheck

Category

Error checking and debugging

Pragma equivalent

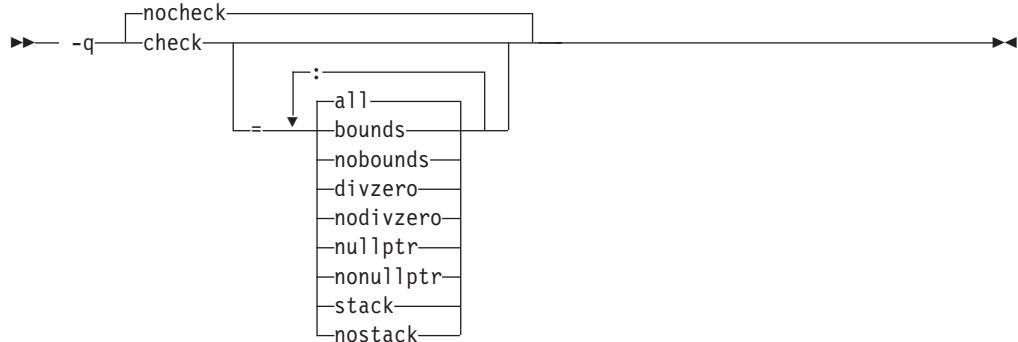
`#pragma options [no]check`

Purpose

Generates code that performs certain types of runtime checking.

If a violation is encountered, a runtime error is raised by sending a SIGTRAP signal to the process (with the exception of the **[no]stack** suboption). Note that the runtime checks may result in slower application execution.

Syntax



Defaults

`-qnocheck`

Parameters

all

Enables all suboptions. For code targeting only the PPU, `-qcheck=all` will not enable `-qcheck=stack`.

bounds | nobounds

Performs runtime checking of addresses for subscripting within an object of known size. The index is checked to ensure that it will result in an address that lies within the bounds of the object's storage. A trap will occur if the address does not lie within the bounds of the object.

This suboption has no effect on accesses to a variable length array.

divzero | nodivzero

Performs runtime checking of integer division. A trap will occur if an attempt is made to divide by zero.

nullptr | nonullptr

Performs runtime checking of addresses contained in pointer variables used to reference storage. The address is checked at the point of use; a trap will occur if the value is less than 512.

stack | nostack (SPU only)

Performs runtime checking for stack overflow. If stack overflow is detected, the program throws an exception by executing a halt instruction. The operating system kernel catches this exception and dumps the memory. The default is **nostack**. **-qcheck** implies **-qcheck=stack**.

Specifying the **-qcheck** option with no suboptions is equivalent to **-qcheck=all**.

Usage

You can specify the **-qcheck** option more than once. The suboption settings are accumulated, but the later suboptions override the earlier ones.

You can use the **all** suboption along with the **no...** form of one or more of the other options as a filter. For example, using:

```
xlc myprogram.c -qcheck=all:nonullptr
```

provides checking for everything except for addresses contained in pointer variables used to reference storage. If you use **all** with the **no...** form of the suboptions, **all** should be the first suboption.

Predefined macros

None.

Examples

The following code example shows the effect of **-qcheck=nullptr:bounds**:

```
void func1(int* p) {
    *p = 42;          /* Traps if p is a null pointer */
}

void func2(int i) {
    int array[10];
    array[i] = 42;    /* Traps if i is outside range 0 - 9 */
}
```

The following code example shows the effect of **-qcheck=divzero**:

```
void func3(int a, int b) {
    a / b;           /* Traps if b=0 */
}
```

-qcinc (C++ only)**Category**

Input control

Pragma equivalent

None.

Purpose

Places an extern "C" { } wrapper around the contents of include files located in a specified directory.

Syntax

►► — -q —  —►►

Defaults

-qnocinc

Parameters

directory_path

The directory where the include files to be wrapped with an extern "C" linkage specifier are located.

Predefined macros

None.

Examples

Assume your application myprogram.C includes header file foo.h, which is located in directory /usr/tmp and contains the following code:

```
int foo();
```

Compiling your application with:

```
ppuxlc++ myprogram.C -qcinc=/usr/tmp
```

will include header file foo.h into your application as:

```
extern "C" {  
int foo();  
}
```

-qcommon (PPU only)

Category

Object code control

Pragma equivalent

None.

Purpose



Controls where uninitialized global variables are allocated.

When **-qcommon** is in effect, uninitialized global variables are allocated in the common section of the object file. When **-qnocommon** is in effect, uninitialized global variables are initialized to zero and allocated in the data section of the object file.

Syntax



Defaults

-  **-qcommon** except when **-qmkshrobj** is specified; **-qnocommon** when **-qmkshrobj** is specified.
-  **-qnocommon**

Usage

This option does not affect static or automatic variables, or the declaration of structure or union members.

This option is overridden by the `common|nocommon` and `section variable` attributes. See "The common and nocommon variable attribute" and "The section variable attribute" in the *XL C/C++ Language Reference*.

Predefined macros

None.

Examples

In the following declaration, where `a` and `b` are global variables:

```
int a, b;
```

Compiling with **-qcommon** produces the equivalent of the following assembly code:

```
.comm _a,4
.comm _b,4
```

Compiling with **-qnocommon** produces the equivalent of the following assembly code:

```
.globl _a
.data
.zerofill __DATA, __common, _a, 4, 2
.globl _b
.data
.zerofill __DATA, __common, _b, 4, 2
```

Related information

- "`-qmkshrobj` (PPU only)" on page 176
- "The common and nocommon variable attribute" in the *XL C/C++ Language Reference*
- "The section variable attribute" in the *XL C/C++ Language Reference*

-qcompact

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]compact

Purpose

Avoids optimizations that increase code size.

Code size is reduced by inhibiting optimizations that replicate or expand code inline, such as inlining or loop unrolling. Execution time may increase.

Syntax

►► -q nocompact
compact _____ ►►

Defaults

-qnocompact

Usage

This option only has an effect when specified with an optimization option.

Predefined macros

`__OPTIMIZE_SIZE__` is predefined to 1 when **-qcompact** and an optimization level are in effect. Otherwise, it is undefined.

Examples

To compile `myprogram.c`, instructing the compiler to reduce code size whenever possible, enter:

invocation `myprogram.c -O -qcompact`

-qcomplexgccincl

Category

Compiler customization

Pragma equivalent

None.

Purpose

Specifies whether to use GCC parameter-passing conventions for complex data types (equivalent to enabling **-qfloat=complexgcc**) for selected include files only.

When **-qcomplexgccincl** is in effect, the compiler internally wraps **#pragma complexgcc(on)** and **#pragma complexgcc(pop)** directives around the files located in specified directories. When **-qnocomplexgccincl** is in effect, include files found in the specified directories are not wrapped by these directives.

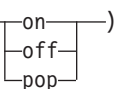
You can also use the pragma directives to enable or disable GCC parameter-passing conventions for complex data types for selected files or sections of code.

Syntax

Option syntax

►► — **-q**  **=** *directory_path* ►►

Pragma syntax

►► **#pragma complexgcc** () ►►

Defaults

By default, files located in the standard directories for the XL C/C++ and GCC header files are wrapped with **#pragma complexgcc** directives. For a list of these, see “Directory search sequence for include files” on page 12.

Parameters

directory_path (option only)

The directory in which the include files to be wrapped with **#pragma complexgcc** directives are located. If you do not specify a *directory_path*, the compiler assumes the default directories listed above.

on (pragma only)

Sets **-qfloat=gccomplex** for the code that follows it. This instructs the compiler to use the GCC conventions for passing and returning parameters of complex type, by using general purpose registers.

off (pragma only)

Sets **-qfloat=nogcccomplex** for the code that follows it. This instructs the compiler to use AIX® conventions for passing and returning parameters of complex type, by using floating-point registers.

pop (pragma only)

Discards the current pragma setting and reverts to the setting specified by the previous pragma directive. If no previous pragma was specified, reverts to the command-line or default option setting.

Usage

The current setting of the pragma affects only functions declared or defined while the setting is in effect. It does not affect other functions.

Calling functions through pointers to functions will always use the convention set by the **-qfloat=[no]complexgcc** command-line option in effect. An error will result

if you mix and match functions that pass complex values by value or return complex values. For example, assume the following code is compiled with **-qfloat=nocomplexgcc**:

```
#pragma complexgcc(on)
void p (_Complex double x) {}

#pragma complexgcc(pop)
typedef void (*fcnptr) (_Complex double);

int main() {
    fcnptr ptr = p; /* error: function pointer is -qfloat=nocomplexgcc;
                    function is -qfloat=complexgcc */
}
```

Predefined macros

None.

Related information

- “-qfloat” on page 98

-qcpluscmt (C only)

Category

Language element control

Pragma equivalent

None.

Purpose

Enables recognition of C++-style comments in C source files.

Syntax

► — -q cpluscmt
nocpluscmt —►

Defaults

- **-qcpluscmt** when the **xlC** or **c99** and related invocations are used, or when the **stdc99** | **extc99** language level is in effect.
- **-qnocpluscmt** for all other invocation commands and language levels.

Predefined macros

__C99_CPLUSCMT is predefined to 1 when **-qcpluscmt** is in effect; otherwise, it is undefined.

Examples

To compile `myprogram.c` so that C++ comments are recognized as comments, enter:
invocation `myprogram.c -qcpluscmt`

Note that `//` comments are *not* part of C89. The result of the following valid C89 program will be incorrect:

```
main() {  
    int i = 2;  
    printf("%i\n", i /* 2 */  
           + 1);  
}
```

The correct answer is 2 (2 divided by 1). When **-qcpluscmt** is in effect (as it is by default), the result is 3 (2 plus 1).

Related information

- “-C, -C!” on page 66
- “-qlanglvl” on page 148
- “Comments” in the *XL C/C++ Language Reference*

-qcrt

Category

Linking

Pragma equivalent

None.

Purpose

Specifies whether system startup files are to be linked.

When **-qcrt** is in effect, the system startup routines are automatically linked. When **-qnocrt** is in effect, the system startup files are not used at link time; only the files specified on the command line with the **-l** flag will be linked.

This option can be used in system programming to disable the automatic linking of the startup routines provided by the operating system.

Syntax

►► — -q  —►►

Defaults

-qcrt

Predefined macros

None.

Related information

- “-qlib” on page 161

-qc_stdinc (C only)

Category

Compiler customization

Pragma equivalent

None.

Purpose

Changes the standard search location for the XL C header files.

Syntax

```
» -qc_stdinc= "directory_path" «
```

Defaults

By default, the compiler searches the directory specified in the configuration file for the XL C header files (this is normally `/opt/ibmcmp/xlc/cbe/10.1/include/`).

Parameters

directory_path

The path for the directory where the compiler should search for the XL C header files. The *directory_path* can be a relative or absolute path. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the XL C headers, you use a configuration file to do so; see “Directory search sequence for include files” on page 12 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the `-qnostdinc` option is in effect.

Predefined macros

None.

Examples

To override the default search path for the XL C headers with `mypath/headers1` and `mypath/headers2`, enter:

```
invocation myprogram.c -qc_stdinc=mypath/headers1:mypath/headers2
```

Related information

- “-qgcc_c_stdinc (C only)” on page 109
- “-qstdinc” on page 225
- “-qinclude” on page 122
- “Directory search sequence for include files” on page 12
- “Specifying compiler options in a configuration file” on page 8

-qcpp_stdinc (C++ only)

Category

Compiler customization

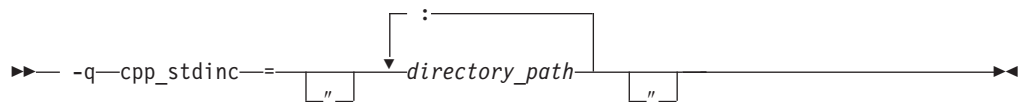
Pragma equivalent

None.

Purpose

Changes the standard search location for the XL C++ header files.

Syntax



Defaults

By default, the compiler searches the directory specified in the configuration file for the XL C++ header files (this is normally `/opt/ibmcmp/xlc/cbe/10.1/include/`).

Parameters

directory_path

The path for the directory where the compiler should search for the XL C++ header files. The *directory_path* can be a relative or absolute path. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the XL C++ headers, you use a configuration file to do so; see “Directory search sequence for include files” on page 12 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the `-qnostdinc` option is in effect.

Predefined macros

None.

Examples

To override the default search path for the XL C++ headers with mypath/headers1 and mypath/headers2, enter:

```
ppuxlc++ myprogram.C -qcpp_stdinc=mypath/headers1:mypath/headers2
```

Related information

- “-qgcc_cpp_stdinc (C++ only)” on page 110
- “-qstdinc” on page 225
- “-qinclude” on page 122
- “Directory search sequence for include files” on page 12
- “Specifying compiler options in a configuration file” on page 8

-D

Category

Language element control

Pragma equivalent

None.

Purpose

Defines a macro as in a #define preprocessor directive.

Syntax

►► -D *name* [= *definition*] ◀◀

Defaults

Not applicable.

Parameters

name

The macro you want to define. -D*name* is equivalent to #define *name*. For example, -DCOUNT is equivalent to #define COUNT.

definition

The value to be assigned to *name*. -D*name*=*definition* is equivalent to #define *name definition*. For example, -DCOUNT=100 is equivalent to #define COUNT 100.

Usage

Using the #define directive to define a macro name already defined by the -D option will result in an error condition.

The **-Uname** option, which is used to undefine macros defined by the **-D** option, has a higher precedence than the **-Dname** option.

Predefined macros

The compiler configuration file uses the **-D** option to predefine several macro names for specific invocation commands. For details, see the configuration file for your system.

Examples

To specify that all instances of the name `COUNT` be replaced by `100` in `myprogram.c`, enter:

```
invocation myprogram.c -DCOUNT=100
```

Related information

- “-U” on page 251
- Chapter 5, “Compiler predefined macros,” on page 325

-qdataimported, -qdatalocal, -qtocdata (PPU only)

Category

Optimization and tuning

Pragma equivalent

None.

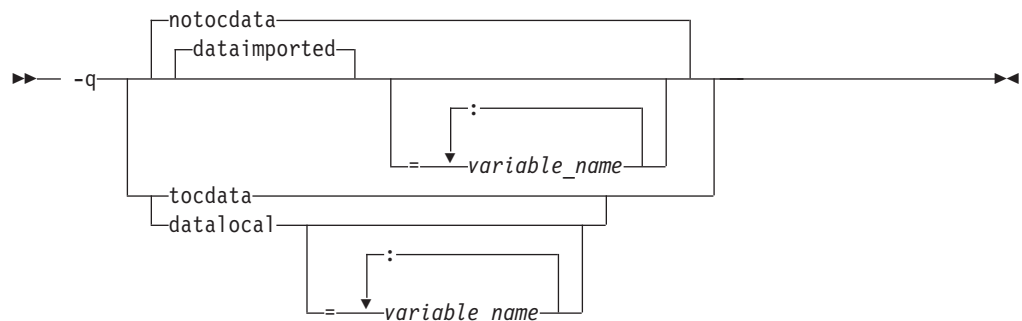
Purpose

Marks data as local or imported in 64-bit compilations.

Local variables are statically bound with the functions that use them. You can use the **-qdatalocal** option to name variables that the compiler can assume are local. Alternatively, you can use the **-qtocdata** option to instruct the compiler to assume all variables are local.

Imported variables are dynamically bound with a shared portion of a library. You can use the **-qdataimported** option to name variables that the compiler can assume are imported. Alternatively, you can use the **-qnotocdata** option to instruct the compiler to assume all variables are imported.

Syntax



Defaults

-qdataimported or **-qnotocdata**: The compiler assumes all variables are imported.

Parameters

variable_name

The name of a variable that the compiler should assume is local or imported (depending on the option specified).

▶ **C++** Names must be specified using their mangled names. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file. (See also "Name mangling" in the *XL C/C++ Language Reference* for details on using the extern "C" linkage specifier on declarations to prevent name mangling.)

Specifying **-qdataimported** without any *variable_name* is equivalent to **-qnotocdata**: all variables are assumed to be imported. Specifying **-qdatalocal** without any *variable_name* is equivalent to **-qtocdata**: all variables are assumed to be local.

Usage

These options apply to 64-bit compilations only.

If any variables that are marked as local are actually imported, incorrect code may be generated and performance may decrease.

If you specify any of these options with no variables, the last option specified is used. If you specify the same variable name on more than one option specification, the last one is used.

Predefined macros

None.

Related information

- “-qprocimported, -qprocllocal, -qprocunknown (PPU only)” on page 195

-qdbxextra (C only)

Category

Error checking and debugging

Pragma equivalent

#pragma options dbxextra

Purpose

When used with the **-g** option, specifies that debugging information is generated for unreferenced typedef declarations, struct, union, and enum type definitions.

To minimize the size of object and executable files, the compiler only includes information for typedef declarations, struct, union, and enum type definitions that are referenced by the program. When you specify the **-qdbxextra** option,

debugging information is included in the symbol table of the object file. This option is equivalent to the **-qsyntab=unref** option.

Syntax

►► — -q — nodbxextra
dbxextra —►►

Defaults

-qnodbxextra: Unreferenced typedef declarations, struct, union, and enum type definitions are not included in the symbol table of the object file.

Usage

Using **-qdbxextra** may make your object and executable files larger.

Predefined macros

None.

Examples

To compile `myprogram.c` so that unreferenced typedef, structure, union, and enumeration declarations are included in the symbol table for use with a debugger, enter:

invocation `myprogram.c -g -qdbxextra`

Related information

- “-qfullpath” on page 106
- “-qlinedebug” on page 163
- “-g” on page 108
- “#pragma options” on page 294
- “-qsyntab (C only)” on page 234

-qdigraph

Category

Language element control

Pragma equivalent

#pragma options [no]digraph



Purpose

Enables recognition of digraph key combinations or keywords to represent characters not found on some keyboards.

Syntax

►► — -q — digraph
nodigraph —►►

Defaults

-  **-qdigraph** when the `extc89` | `extended` | `extc99` | `stdc99` language level is in effect. **-qnodigraph** for all other language levels.
-  **-qdigraph**

Usage

A digraph is a keyword or combination of keys that lets you produce a character that is not available on all keyboards. For details on digraphs, see "Digraph characters" in the *XL C/C++ Language Reference*.

Predefined macros

`__DIGRAPHS__` is predefined to 1 when **-qdigraph** is in effect; otherwise it is not defined.

Examples

To disable digraph character sequences when compiling your program, enter:

```
invocation myprogram.c -qnodigraph
```

Related information

- `"-qlanglvl"` on page 148
- `"-qtrigraph"` on page 249

-qdirectstorage (PPU only)

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.

Syntax

►► — -q  —►►

Defaults

`-qnodirectstorage`

Usage

Use this option with discretion. It is intended for programmers who know how the memory and cache blocks work, and how to tune their applications for optimal performance. To ensure that your application will execute correctly on all

implementations, you should assume that separate instruction and data caches exist and program your application accordingly.

-qdollar

Category

Language element control

Pragma equivalent

#pragma options [no]dollar

Purpose

Allows the dollar-sign (\$) symbol to be used in the names of identifiers.

When **dollar** is in effect, the dollar symbol \$ in an identifier is treated as a base character.

Syntax

►► — -q ————— ◄◄
 └── nodollar ──┘
 └── dollar ───┘

Defaults

-qnodollar

Usage

If **nodollar** and the **ucs** language level are both in effect, the dollar symbol is treated as an extended character and translated into \u0024.

Predefined macros

None.

Examples

To compile `myprogram.c` so that \$ is allowed in identifiers in the program, enter:

invocation `myprogram.c -qdollar`

Related information

- “-qlanglvl” on page 148

-qdump_class_hierarchy (C++ only)

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Dumps a representation of the hierarchy and virtual function table layout of each class object to a file.

Syntax

►► — `-q—dump_class_hierarchy—` ◄◄

Defaults

Not applicable.

Usage

The output file name consists of the source file name appended with a `.class` suffix.

Predefined macros

None.

Examples

To compile `myprogram.C` to produce a file named `myprogram.C.class` containing the class hierarchy information, enter:

```
ppuxlc++ myprogram.C -qdump_class_hierarchy
```

-e

Category

Linking

Pragma equivalent

None.

Purpose

When used together with the `-qmkshrobj`, specifies an entry point for a shared object.

Syntax

►► — `-e` `noentry`
`name` ◄◄

Defaults

`-e=noentry`

Parameters

name

The name of the entry point for the shared executable.

Usage

When linking object files, it is recommended that you *do not* use the **-e** option. The default entry point of the executable output is `__start`. Changing this label with the **-e** flag can cause erratic results.

This option is used only together with the **-qmkshrobj** option. See the description for the “**-qmkshrobj** (PPU only)” on page 176 for more information.

Predefined macros

None.

Related information

- “**-qmkshrobj** (PPU only)” on page 176

-E

Category

Output control

Pragma equivalent

None.

Purpose

Preprocesses the source files named in the compiler invocation, without compiling, and writes the output to the standard output.

Syntax

►► -E ◀◀

Defaults

By the default, source files are preprocessed, compiled, and linked to produce an executable file.

Usage

The **-E** option accepts any file name. Source files with unrecognized file name suffixes are treated and preprocessed as C files, and no error message is generated.

Unless **-qnoppline** is specified, `#line` directives are generated to preserve the source coordinates of the tokens. Continuation sequences are preserved.

Unless **-C** is specified, comments are replaced in the preprocessed output by a single space character. New lines and `#line` directives are issued for comments that span multiple source lines.

The **-E** option overrides the **-P**, **-o**, and **-qsyntaxonly** options.

Predefined macros

None.

Examples

To compile myprogram.c and send the preprocessed source to standard output, enter:

```
invocation myprogram.c -E
```

If myprogram.c has a code fragment such as:

```
#define SUM(x,y) (x + y)
int a ;
#define mm 1    /* This is a comment in a
                  preprocessor directive */
int b ;         /* This is another comment across
                  two lines */
int c ;
                  /* Another comment */
c = SUM(a, /* Comment in a macro function argument*/
        b) ;
```

the output will be:

```
#line 2 "myprogram.c"
int a ;
#line 5
int b ;

int c ;

c = a + b ;
```

Related information

- “-qpplne” on page 191
- “-C, -C!” on page 66
- “-P” on page 184
- “-qsyntaxonly (C only)” on page 235

-qea32, -qea64 (SPU only)

Category

Language element control

Pragma equivalent

None.

Purpose

Compiles SPU code that uses effective address space support for either a 32-bit or 64-bit PPE address space.

Syntax

►► — -q  —►►

Defaults

-qea32

Usage

Compile with **-qea32** if you are linking your SPU program to a 32-bit PPU object, and with **-qea64** if you are linking your SPU program to a 64-bit PPU object. Only one of these options should be used at a time. If an SPU object is compiled using **-qea32** and then linked to a 64-bit PPU object, the behavior of the overall program is undefined; this is an invalid use of effective address space support.

These options are valid only for SPU invocation commands; for example, **spuxlc**. They are supported by the C compiler as both compile-time and link-time options. However, they are accepted by the C++ compiler only as link-time options.

Predefined macros

- `__EA32__` is predefined when **-qea32** is specified (C only).
- `__EA64__` is predefined when **-qea64** is specified (C only).

Examples

To compile program `myprogram.c` so that it uses effective address space support, enter:

```
spuxlc myprogram.c -qea64
```

Related information

- "Accessing PPU variables from SPU programs" in the *XL C/C++ Optimization and Programming Guide*
- "The `__ea` type qualifier" in the *XL C/C++ Language Reference*

-qeh (C++ only) (PPU only)

Category

Object code control

Pragma equivalent

None.

Purpose

Controls whether exception handling is enabled in the module being compiled.

When **-qeh** is in effect, exception handling is enabled. If your program does not use C++ structured exception handling, you can compile with **-qnoeh** to prevent generation of code that is not needed by your application.

Syntax

►► -q eh
noeh ►►

Defaults

-qeh

Usage

Specifying **-qeh** also implies **-qrtti**. If **-qeh** is specified together with **-qnortti**, RTTI information will still be generated as needed.

Predefined macros

`__EXCEPTIONS` is predefined to 1 when **-qeh** is in effect; otherwise, it is undefined.

Related information

- “-qrtti (C++ only) (PPU only)” on page 207

-qenum

Category

Floating-point and integer control

Pragma equivalent

`#pragma options enum`, `#pragma enum`

Purpose

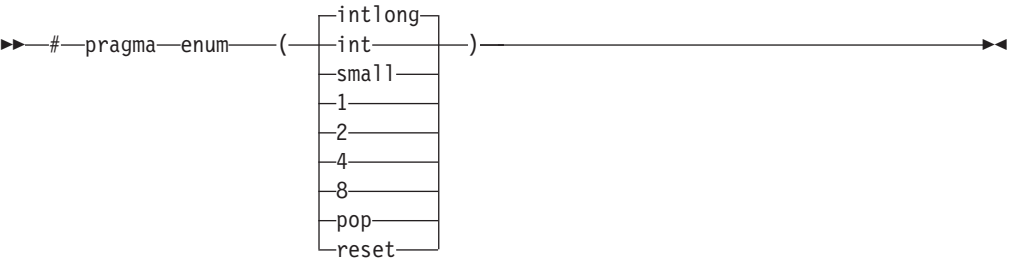
Specifies the amount of storage occupied by enumerations.

Syntax

Option syntax




Pragma syntax



Defaults

-qenum=intlong

Parameters

- 1 Specifies that enumerations occupy 1 byte of storage, are of type char if the range of enumeration values falls within the limits of signed char, and unsigned char otherwise.
- 2 Specifies that enumerations occupy 2 bytes of storage, are of type short if the range of enumeration values falls within the limits of signed short, and unsigned short otherwise.  Values cannot exceed the range of signed int.
- 4 | **int**
Specifies that enumerations occupy 4 bytes of storage, are of type int if the range of enumeration values falls within the limits of signed int, and unsigned int otherwise.
- 8 Specifies that enumerations occupy 8 bytes of storage. In 32-bit compilation mode, the enumeration is of type long long if the range of enumeration values falls within the limits of signed long long, and unsigned long long otherwise. In 64-bit compilation mode (on the PPU only), the enumeration is of type long if the range of enumeration values falls within the limits of signed long, and unsigned long otherwise.

intlong

Specifies that enumerations occupy 8 bytes of storage if the range of values in the enumeration exceeds the limit for int. If the range of values in the enumeration does not exceed the limit for int, the enumeration will occupy 4 bytes of storage and is represented by int.

small

Specifies that enumerations occupy the smallest amount of space (1, 2, 4, or 8 bytes of storage) that can accurately represent the range of values in the enumeration. Signage is unsigned, unless the range of values includes negative values. If an 8-byte enum results, the actual enumeration type used is dependent on compilation mode.

pop | reset (pragma only)

Discards the current pragma setting and reverts to the setting specified by the previous pragma directive. If no previous pragma was specified, reverts to the command-line or default option setting.

Usage

The tables that follow show the priority for selecting a predefined type. The table also shows the predefined type, the maximum range of enum constants for the corresponding predefined type, and the amount of storage that is required for that predefined type, that is, the value that the sizeof operator would yield when applied to the minimum-sized enum. All types are signed unless otherwise noted.

Table 21. Enumeration sizes and types

	enum=1		enum=2		enum=4		enum=8			
							32-bit compilation mode		64-bit compilation mode (PPU only)	
Range	var	const	var	const	var	const	var	const	var	const
0..127	char	int	short	int	int	int	long long	long long	long	long

Table 21. Enumeration sizes and types (continued)


-128..127	char	int	short	int	int	int	long long	long long	long	long
0..255	unsigned char	int	short	int	int	int	long long	long long	long	long
0..32767	ERROR ¹	int	short	int	int	int	long long	long long	long	long
-32768..32767	ERROR ¹	int	short	int	int	int	long long	long long	long	long
0..65535	ERROR ¹	int	unsigned short	int	int	int	long long	long long	long	long
0..2147483647	ERROR ¹	int	ERROR ¹	int	int	int	long long	long long	long	long
-(2147483647+1).. 2147483647	ERROR ¹	int	ERROR ¹	int	int	int	long long	long long	long	long
0..4294967295	ERROR ¹	unsigned int	ERROR ¹	unsigned int	unsigned int	unsigned int	long long	long long	long	long
0..(2 ⁶³ -1)	ERROR ¹	long ²	ERROR ¹	long ²	ERROR ¹	long ²	long long ²	long long ²	long ²	long ²
-2 ⁶³ ..(2 ⁶³ -1)	ERROR ¹	long ²	ERROR ¹	long ²	ERROR ¹	long ²	long long ²	long long ²	long ²	long ²
0..2 ⁶⁴	ERROR ¹	unsigned long ²	ERROR ¹	unsigned long ²	ERROR ¹	unsigned long ²	unsigned long long ²	unsigned long long ²	unsigned long ²	unsigned long ²

	enum=int		enum=intlong				enum=small			
			32-bit compilation mode		64-bit compilation mode		32-bit compilation mode		64-bit compilation mode (PPU only)	
Range	var	const	var	const	var	const	var	const	var	const
0..127	int	int	int	int	int	int	unsigned char	int	unsigned char	int
-128..127	int	int	int	int	int	int	signed char	int	signed char	int
0..255	int	int	int	int	int	int	unsigned char	int	unsigned char	int
0..32767	int	int	int	int	int	int	unsigned short	int	unsigned short	int
-32768..32767	int	int	int	int	int	int	short	int	short	int
0..65535	int	int	int	int	int	int	unsigned short	int	unsigned short	int
0..2147483647	int	int	int	int	int	int	unsigned int	unsigned int	unsigned int	unsigned int
-(2147483647+1).. 2147483647	int	int	int	int	int	int	int	int	int	int
0..4294967295	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int
0..(2 ⁶³ -1)	ERR ²	ERR ²	long long ²	long long ²	long ²	long ²	unsigned long long ²	unsigned long long ²	unsigned long ²	unsigned long ²
-2 ⁶³ ..(2 ⁶³ -1)	ERR ²	ERR ²	long long ²	long long ²	long ²	long ²	long long ²	long long ²	long ²	long ²
0..2 ⁶⁴	ERR ²	ERR ²	unsigned long long ²	unsigned long long ²	unsigned long ²	unsigned long ²	unsigned long long ²	unsigned long long ²	unsigned long ²	unsigned long ²

Note:

1. These enumerations are too large for the **-qenum=1|2|48** settings. A Severe error is issued and compilation stops. To correct this condition, you should

reduce the range of the enumerations, choose a larger **-qenum** setting, or choose a dynamic **-qenum** setting, such as **small** or **intlong**.

2.  Enumeration types must not exceed the range of int when compiling C applications to ISO C 1989 and ISO C 1999 Standards. When the **stdc89** | **stdc99** language level in effect, the compiler will behave as follows if the value of an enumeration exceeds the range of int:
 - a. If **-qenum=int** is in effect, a severe error message is issued and compilation stops.
 - b. For all other settings of **-qenum**, an informational message is issued and compilation continues.

The **#pragma enum** directive must be precede the declaration of enum variables that follow; any directives that occur within a declaration are ignored and diagnosed with a warning.

For each **#pragma enum** directive that you put in a source file, it is good practice to have a corresponding **#pragma enum=reset** before the end of that file. This should prevent one file from potentially changing the setting of another file that includes it.

Examples

If the following fragment is compiled with the **enum=small** option:

```
enum e_tag {a, b, c} e_var;
```

the range of enumeration constants is 0 through 2. This range falls within all of the ranges described in the table above. Based on priority, the compiler uses predefined type unsigned char.

If the following fragment is compiled with the **enum=small** option:

```
enum e_tag {a=-129, b, c} e_var;
```

the range of enumeration constants is -129 through -127. This range only falls within the ranges of short (signed short) and int (signed int). Because short (signed short) is smaller, it will be used to represent the enum.

The following code segment generates a warning and the second occurrence of the **enum** pragma is ignored:

```
#pragma enum=small
enum e_tag {
    a,
    b,
    #pragma enum=int /* error: cannot be within a declaration */
    c
} e_var;
#pragma enum=reset /* second reset isn't required */
```

The range of enum constants must fall within the range of either unsigned int or int (signed int). For example, the following code segments contain errors:

```
#pragma enum=small
enum e_tag { a=-1,
             b=2147483648 /* error: larger than maximum int */
             } e_var;
#pragma options enum=reset
```

Predefined macros

None.

-qenablevmx (PPU only)

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables generation of vector instructions.

These instructions can offer higher performance when used with algorithmic-intensive tasks such as multimedia applications.

Syntax

►► -q enablevmx
noenablevmx _____ ◄◄

Defaults

-qenablevmx for all supported **-qarch** values.

-qnoenablevmx, otherwise.

Usage

If **-qnoenablevmx** is in effect, **-qaltivec** and **-qhot=simd** cannot be used.

Predefined macros

None.

Related information

- “-qaltivec” on page 56
- “-qarch” on page 57
- “-qhot” on page 115

-F

Category

Compiler customization

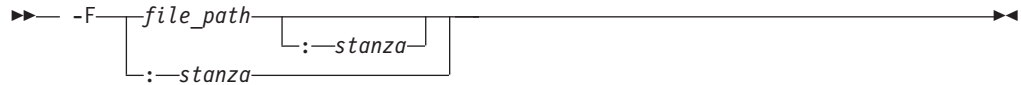
Pragma equivalent

None.

Purpose

Names an alternative configuration file or stanza for the compiler.

Syntax



Defaults

By default, the compiler uses the configuration file that is configured at installation time, and uses the stanza defined in that file for the invocation command currently being used.

Parameters

file_path

The full path name of the alternate compiler configuration file to use.

stanza

The name of the configuration file stanza to use for compilation. This directs the compiler to use the entries under that *stanza* regardless of the invocation command being used. For example, if you are compiling with **ppuxlc** or **spuxlc**, but you specify the **c99** stanza, the compiler will use all the settings specified in the **c99** stanza.

Usage

Note that any file names or stanzas that you specify with the **-F** option override the defaults specified in the system configuration file. If you have specified a custom configuration file with the `XLC_USR_CONFIG` environment variable, that file is processed before the one specified by the **-F** option.

The **-B**, **-t**, and **-W** options override the **-F** option.

Predefined macros

None.

Examples

To compile `myprogram.c` using a stanza called `debug` that you have added to the default configuration file, enter:

```
invocation myprogram.c -F:debug
```

To compile `myprogram.c` using a configuration file called `/usr/tmp/myconfig.cfg`, enter:

```
invocation myprogram.c -F/usr/tmp/myconfig.cfg
```

To compile `myprogram.c` using the stanza `c99` you have created in a configuration file called `/usr/tmp/myconfig.cfg`, enter:

```
invocation myprogram.c -F/usr/tmp/myconfig.cfg:c99
```

Related information

- “Using custom compiler configuration files” on page 24
- “-B” on page 63
- “-t” on page 236
- “-W” on page 262
- “Specifying compiler options in a configuration file” on page 8
- “Compile-time and link-time environment variables” on page 22

-qfdpr

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Provides object files with information that the IBM Feedback Directed Program Restructuring (FDPR) performance-tuning utility needs to optimize the resulting executable file.

When **-qfdpr** is in effect, optimization data is stored in the object file.

Syntax

►► — -q nofdpr
fdpr —►►

Defaults

-qnofdpr

Usage

For best results, use **-qfdpr** for all object files in a program; FDPR will perform optimizations only on the files compiled with **-qfdpr**, and not library code, even if it is statically linked.

The optimizations that the FDPR utility performs are similar to those that the **-qpdf** option performs.

The FDPR performance-tuning utility has its own set of restrictions, and it is not guaranteed to speed up all programs or produce executables that produce exactly the same results as the original programs.

Predefined macros

None.

Examples

To compile `myprogram.c` so it includes data required by the FDPR utility, enter:

invocation `myprogram.c -qf DPR`

Related information

- “-qpdf1, -qpdf2 (PPU only)” on page 187

-qflag

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options flag, “#pragma report (C++ only)” on page 304

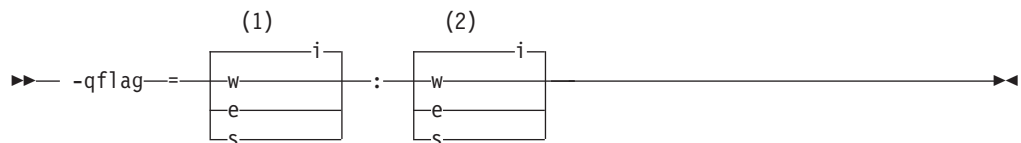
Purpose

Limits the diagnostic messages to those of a specified severity level or higher.

The messages are written to standard output and, optionally, to the listing file if one is generated.

Syntax

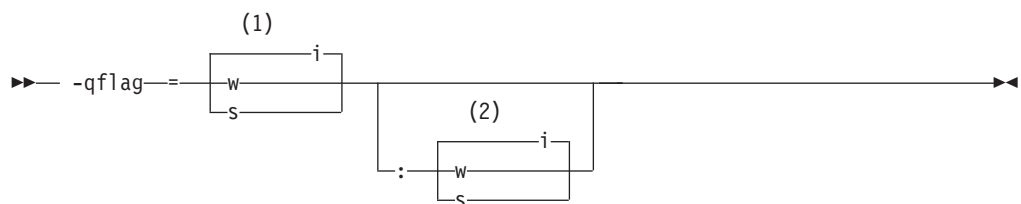
-qflag syntax – C



Notes:

- 1 Minimum severity level of messages reported in listing
- 2 Minimum severity level of messages reported on terminal

-qflag syntax – C++






Notes:

- 1 Minimum severity level of messages reported in listing
- 2 Minimum severity level of messages reported on terminal


Defaults


`-qflag=i` : i, which shows all compiler messages

Parameters

- i** Specifies that all diagnostic messages are to display: warning, error and informational messages. Informational messages (I) are of the lowest severity.
- w** Specifies that warning (W) and all types of error messages are to display.
-  **e** Specifies that only error (E), severe error (S), and unrecoverable error (U) messages are to display.
- s**  Specifies that only severe error (S) and unrecoverable error (U) messages are to display.  Specifies that only severe error (S) messages are to display.

Usage

 You must specify a minimum message severity level for both listing and terminal reporting.

 You must specify a minimum message severity level for the listing. If you do not specify a suboption for the terminal, the compiler assumes the same severity as for the listing.

Note that using `-qflag` does not enable the classes of informational message controlled by the `-qinfo` option; see `-qinfo` for more information.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the listing shows all messages that were generated and your workstation displays only error and higher messages (with their associated information messages to aid in fixing the errors), enter:

invocation `myprogram.c -qflag=i:e`

Related information

- “`-qinfo`” on page 124
- “`-w`” on page 261
- “Compiler messages” on page 15

`-qfloat`

Category

Floating-point and integer control

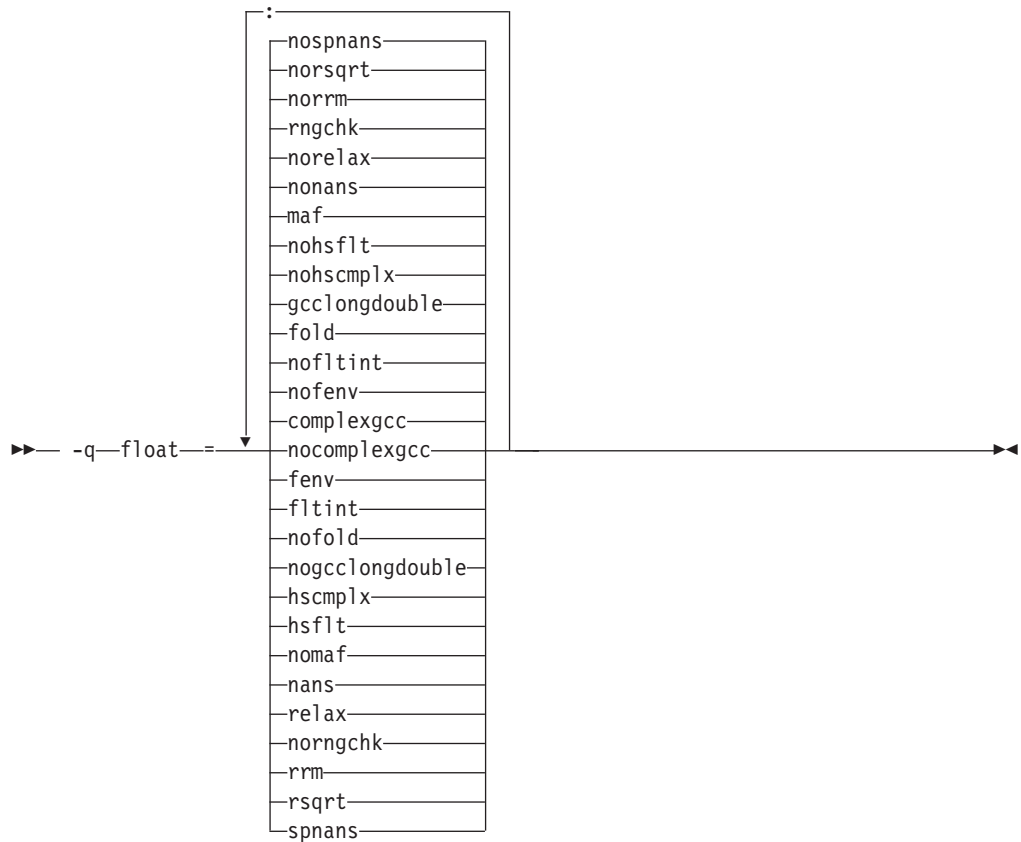
Pragma equivalent

`#pragma options float`

Purpose

Selects different strategies for speeding up or improving the accuracy of floating-point calculations.

Syntax



Defaults

- `-qfloat=complexgcc:nofenv:nofltint:fold:gcclongdouble: nohscmplx:nohsflt:maf:nonans:norelax:rngchk:norm:norsqrt:nospnans` when compiled with **ppu** or **cbe** prefixed invocation commands
- `-qfloat=complexgcc:nofenv:nofltint:fold:gcclongdouble:nohscmplx:nohsflt:maf: nans:norelax:rngchk:norm:norsqrt:spnans` when compiled with **spu** prefixed invocation commands
- `-qfloat=fltint:rsqrt:norngchk` when `-qnostrict`, `-qstrict=nooperationprecision:noexceptions`, or `-O3` or higher optimization level is in effect.
- `-qfloat=nocomplexgcc` when 64-bit mode is enabled.

Parameters

complexgcc | nocomplexgcc (PPU only)

Specifies whether GCC conventions for passing or returning complex numbers are to be used. **complexgcc** preserves compatibility with GCC-compiled code. This suboption does not have any effect if support for complex types is not in effect; see “`-qlanglvl`” on page 148 for details.

fenv | nofenv

Specifies whether the code depends on the hardware environment and whether to suppress optimizations that could cause unexpected results due to this dependency.

Certain floating-point operations rely on the status of Floating-Point Status and Control Register (FPSCR), for example, to control the rounding mode or to detect underflow. In particular, many compiler built-in functions read values directly from the FPSCR.

When **nofenv** is in effect, the compiler assumes that the program does not depend on the hardware environment, and that aggressive compiler optimizations that change the sequence of floating-point operations are allowed. When **fenv** is in effect, such optimizations are suppressed.

You should use **fenv** for any code containing statements that read or set the hardware floating-point environment, to guard against optimizations that could cause unexpected behavior.

Any directives specified in the source code (such as the standard C `FENV_ACCESS` pragma) take precedence over the option setting.

fltint | nofltint (PPU only)

Speeds up floating-point-to-integer conversions by using an inline sequence of code instead of a call to a library function. The library function, which is called when **nofltint** is in effect, checks for floating-point values outside the representable range of integers and returns the minimum or maximum representable integer if passed an out-of-range floating-point value.

If **-qarch** is set to a processor that has an instruction to convert from floating point to integer, that instruction will be used regardless of the **[no]fltint** setting. This conversion also applies to all Power processors in 64-bit mode.

If you compile with **-O3** or higher optimization level, **fltint** is enabled automatically. To disable it, also specify **-qstrict**, **-qstrict=operationprecision**, or **-qstrict=exceptions**.

fold | nofold

Evaluates constant floating-point expressions at compile time, which may yield slightly different results from evaluating them at run time. The compiler always evaluates constant expressions in specification statements, even if you specify **nofold**.

gcclongdouble | nogcclongdouble (PPU only)

Specifies whether the compiler uses GCC-supplied or IBM-supplied library functions for 128-bit long double operations.

gcclongdouble ensures binary compatibility with GCC for mathematical calculations. If this compatibility is not important in your application, you should use **nogcclongdouble** for better performance. This suboption only has an effect when 128-bit long double types are enabled with **-qldbl128**.

Note: Passing results from modules compiled with **nogcclongdouble** to modules compiled with **gcclongdouble** may produce different results for numbers such as Inf, NaN and other rare cases. To avoid such incompatibilities, the compiler provides built-in functions to convert IBM long double types to GCC long double types; see “Floating-point built-in functions” on page 345 for more information.

hscmplx | nohscmplx

Speeds up operations involving complex division and complex absolute value.

This suboption, which provides a subset of the optimizations of the **hsflt** suboption, is preferred for complex calculations.

hsflt | nohsflt

Speeds up calculations by preventing rounding for single-precision expressions and by replacing floating-point division by multiplication with the reciprocal of the divisor. It also uses the same technique as the **fltint** suboption for floating-point-to-integer conversions. **hsflt** implies **hscmplx**.

The **hsflt** suboption overrides the **nans** and **spnans** suboptions.

Note: Use **-qfloat=hsflt** on applications that perform complex division and floating-point conversions where floating-point calculations have known characteristics. In particular, all floating-point results must be within the defined range of representation of single precision. Use with discretion, as this option may produce unexpected results without warning. For complex computations, it is recommended that you use the **hscmplx** suboption (described above), which provides equivalent speed-up without the undesirable results of **hsflt**.

maf | nomaf

Makes floating-point calculations faster and more accurate by using floating-point multiply-add instructions where appropriate. The results may not be exactly equivalent to those from similar calculations performed at compile time or on other types of computers. Negative zero results may be produced. This suboption may affect the precision of floating-point intermediate results. If **-qfloat=nomaf** is specified, no multiply-add instructions will be generated unless they are required for correctness.

nans | nonans (PPU only)

Allows you to use the **-qflttrap=invalid:enable** option to detect and deal with exception conditions that involve signaling NaN (not-a-number) values. Use this suboption only if your program explicitly creates signaling NaN values, because these values never result from other floating-point operations.

relax | norelax

Relaxes strict IEEE conformance slightly for greater speed, typically by removing some trivial floating-point arithmetic operations, such as adds and subtracts involving a zero on the right. These changes are allowed if either **-qstrict=noieee** or **-qfloat=relax** is specified.

rngchk | norngchk

At optimization level **-O3** and above, and without **-qstrict**, controls whether range checking is performed for input arguments for software divide and inlined square root operations. Specifying **norngchk** instructs the compiler to skip range checking, allowing for increased performance where division and square root operations are performed repeatedly within a loop.

Note that with **norngchk** in effect the following restrictions apply:

- The dividend of a division operation must not be +/-INF.
- The divisor of a division operation must not be 0.0, +/- INF, or denormalized values.
- The quotient of dividend and divisor must not be +/-INF.
- The input for a square root operation must not be INF.

If any of these conditions are not met, incorrect results may be produced. For example, if the divisor for a division operation is 0.0 or a denormalized number (absolute value $< 2^{-1022}$ for double precision, and absolute value $< 2^{-126}$

for single precision), NaN, instead of INF, may result; when the divisor is +/- INF, NaN instead of 0.0 may result. If the input is +INF for a sqrt operation, NaN, rather than INF, may result.

norngchk is only allowed when **-qnostrict** is in effect. If **-qstrict**, **-qstrict=infinities**, **-qstrict=operationprecision**, or **-qstrict=exceptions** is in effect, **norngchk** is ignored.

rrm | **norrm** (PPU only)

Prevents floating-point optimizations that require the rounding mode to be the default, round-to-nearest, at run time, by informing the compiler that the floating-point rounding mode may change or is not round-to-nearest at run time. You should use **rrm** if your program changes the runtime rounding mode by any means; otherwise, the program may compute incorrect results.

rsqrt | **norsqrt**

Speeds up some calculations by replacing division by the result of a square root with multiplication by the reciprocal of the square root.

rsqrt has no effect unless **-qignerrno** is also specified; **errno** will *not* be set for any sqrt function calls.

If you compile with **-O3** or higher optimization level, **rsqrt** is enabled automatically. To disable it, also specify **-qstrict**, **-qstrict=nans**, **-qstrict=infinities**, **-qstrict=zerosigns**, or **-qstrict=exceptions**.

spnans | **nospnans** (PPU only)

Generates extra instructions to detect signalling NaN on conversion from single-precision to double-precision.

Usage

Using **-qfloat** suboptions other than the default settings may produce incorrect results in floating-point computations if not all required conditions for a given suboption are met. For this reason, you should only use this option if you are experienced with floating-point calculations involving IEEE floating-point values and can properly assess the possibility of introducing errors in your program. See also "Handling floating point operations" in the *XL C/C++ Optimization and Programming Guide* for more information.

If the **-qstrict** | **-qnostrict** and **float** suboptions conflict, the last setting specified is used.

Predefined macros

None.

Examples

To compile `myprogram.c` so that constant floating point expressions are evaluated at compile time and multiply-add instructions are not generated, enter:

```
invocation myprogram.c -qfloat=fold:nomaf
```

Related information

- **"-qarch"** on page 57
- **"-qcomplexgccincl"** on page 74
- **"-qflttrap (PPU only)"** on page 103
- **"-qstrict"** on page 227

-qflttrap (PPU only)

Category

Error checking and debugging

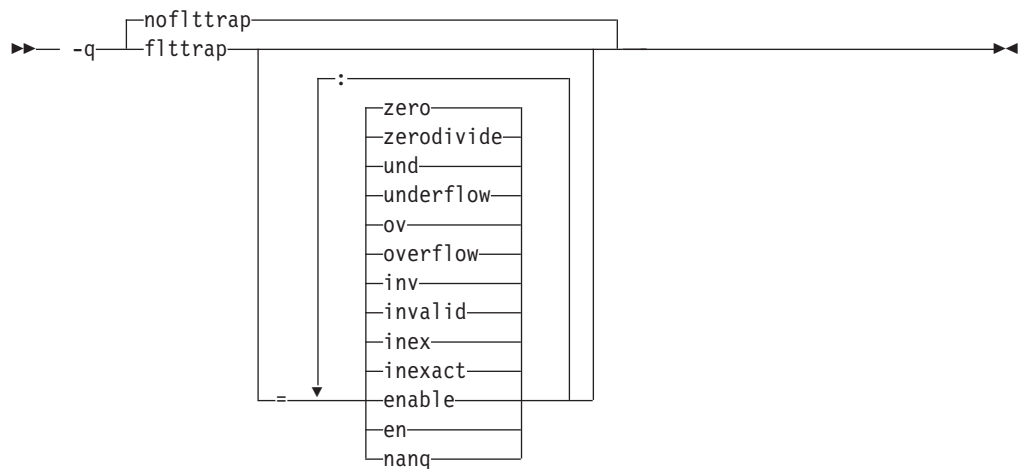
Pragma equivalent

#pragma options [no]flttrap

Purpose

Determines the types of floating-point exception conditions to be detected at run time

Syntax



Defaults

-qnoflttrap

Parameters

enable, en

Enables trapping when the specified exceptions (**overflow**, **underflow**, **zerodivide**, **invalid**, or **inexact**) occur. You must specify this suboption if you want to turn on exception trapping without modifying your source code. If any of the specified exceptions occur, a SIGTRAP or SIGFPE signal is sent to the process with the precise location of the exception.

inexact, inex

Enables the detection of floating-point inexact operations. If a floating-point inexact operation occurs, an inexact operation exception status flag is set in the Floating-Point Status and Control Register (FPSCR).

invalid, inv

Enables the detection of floating-point invalid operations. If a floating-point invalid operation occurs, an invalid operation exception status flag is set in the FPSCR.

nanq

Generates code to detect NaNQ (Not a Number Quiet) and NaNs (Not a

Number Signalling) exceptions before and after each floating point operation, including assignment, and after each call to a function returning a floating-point result to trap if the value is a NaN. Trapping code is generated regardless of whether the **enable** suboption is specified.

overflow, ov

Enables the detection of floating-point overflow. If a floating-point overflow occurs, an overflow exception status flag is set in the FPSCR.

underflow, und

Enables the detection of floating-point underflow. If a floating-point underflow occurs, an underflow exception status flag is set in the FPSCR.

zerodivide, zero

Enables the detection of floating-point division by zero. If a floating-point zero-divide occurs, a zero-divide exception status flag is set in the FPSCR.

Specifying **-qflttrap** option with no suboptions is equivalent to **-qflttrap=overflow : underflow : zerodivide : invalid : inexact**. Exceptions will be detected by the hardware, but trapping is not enabled. Because this default does not include **enable**, it is probably only useful if you already use **fpsets** or similar subroutines in your source.

Usage

It is recommended that you use the **enable** suboption whenever compiling the main program with **-qflttrap**. This ensures that the compiler will generate the code to automatically enable floating-point exception trapping, without requiring that you include calls to the appropriate floating-point exception library functions in your code.

If you specify **-qflttrap** more than once, both with and without suboptions, the **-qflttrap** without suboptions is ignored.

This option is recognized during linking with IPA. Specifying the option at the link step overrides the compile-time setting.

If your program contains signalling NaNs, you should use the **-qfloat=nans** option along with **-qflttrap** to trap any exceptions.

The compiler exhibits behavior as illustrated in the following examples when the **-qflttrap** option is specified together with an optimization option:

- with **-O2**:
 - 1/0 generates a **div0** exception and has a result of infinity
 - 0/0 generates an invalid operation
- with **-O3** or greater:
 - 1/0 generates a **div0** exception and has a result of infinity
 - 0/0 returns zero multiplied by the result of the previous division.

Predefined macros

None.

Examples

When you compile this program:

```
#include <stdio.h>

int main()
{
    float x, y, z;
    x = 5.0;
    y = 0.0;
    z = x / y;
    printf("%f", z);
}
```

with the command:

```
invocation -qf1ttrap=zerodivide:enable divide_by_zero.c
```

the program stops when the division is performed.

The **zerodivide** suboption identifies the type of exception to guard against. The **enable** suboption causes a SIGTRAP or SIGFPE signal to be generated when the exception occurs.

Related information

- “-qfloat” on page 98

-qformat

Category

Error checking and debugging

Pragma equivalent

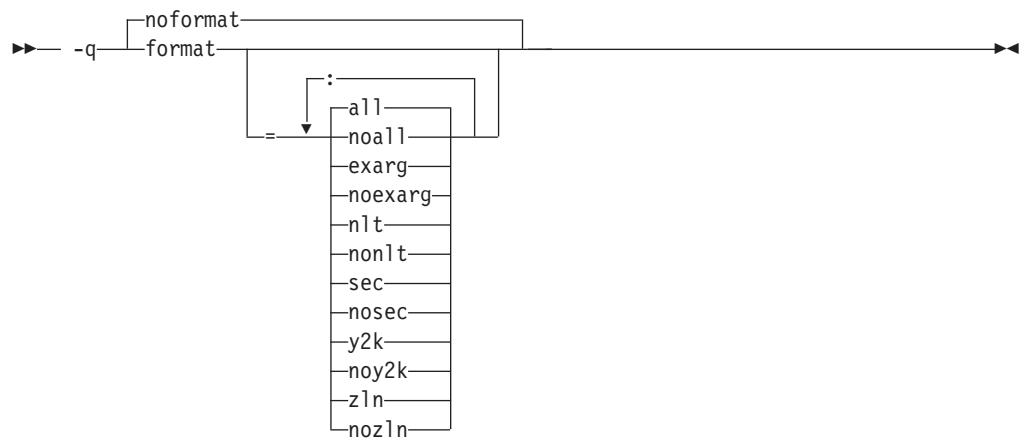
None.

Purpose

Warns of possible problems with string input and output format specifications.

Functions diagnosed are printf, scanf, strftime, strfmon family functions and functions marked with format attributes.

Syntax



Defaults

`-qnoformat`

Parameters

`all` | `noall`

Enables or disables all format diagnostic messages.

`exarg` | `noexarg`

Warns if excess arguments appear in `printf` and `scanf` style function calls.

`nlt` | `nonlt`

Warns if a format string is not a string literal, unless the format function takes its format arguments as a `va_list`.

`sec` | `nosec`

Warns of possible security problems in use of format functions.

`y2k` | `noy2k`

Warns of `strftime` formats that produce a 2-digit year.

`zln` | `nozln`

Warns of zero-length formats.

Specifying `-qformat` with no suboptions is equivalent to `-qformat=all`.

`-qnoformat` is equivalent to `-qformat=noall`.

Predefined macros

None.

Examples

To enable all format string diagnostics, enter either of the following:

invocation `myprogram.c -qformat=all`

invocation `myprogram.c -qformat`

To enable all format diagnostic checking except that for y2k date diagnostics, enter:

invocation `myprogram.c -qformat=all:noy2k`

`-qfullpath`

Category

Error checking and debugging

Pragma equivalent

`#pragma options [no]fullpath`

Purpose

When used with the `-g` or `-qlinedebug` option, this option records the full, or absolute, path names of source and include files in object files compiled with debugging information, so that debugging tools can correctly locate the source files.

When **fullpath** is in effect, the absolute (full) path names of source files are preserved. When **nofullpath** is in effect, the relative path names of source files are preserved.

Syntax

►► — -q — nofullpath
fullpath —►►

Defaults

-qnofullpath

Usage

If your executable file was moved to another directory, the debugger would be unable to find the file unless you provide a search path in the debugger. You can use **fullpath** to ensure that the debugger locates the file successfully.

Predefined macros

None.

Related information

- “-qlinedebug” on page 163
- “-g” on page 108

-qfuncsect

Category

Object code control

Pragma equivalent

#pragma options [no]funcsect

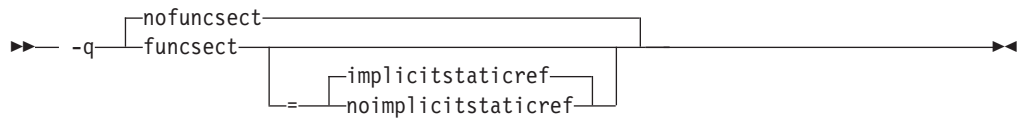
Purpose

Places instructions for each function in a separate object file control section or CSECT.

When **-qfuncsect** is specified the compiler generates references from each function to the static data area, if one exists, in order to ensure that if any function from that object file is included in the final executable, the static data area also is included. This is done to ensure that any static strings or strings from a pragma comment, possibly containing copyright information, are also included in the executable. This can, in some cases, cause code bloat or unresolved symbols at link time.

When **-qnofuncsect** is in effect, each object file consists of a single control section combining all functions defined in the corresponding source file. You can use **-qfuncsect** to place each function in a separate control section.

Syntax



Defaults

-qnofuncsect

Parameters

implicitstaticref | **noimplicitstaticref**

Specifies whether references to the static data section of the object file by functions contained in static variables, virtual function tables, or exception handling tables, are maintained.

When your code contains a **#pragma comment** directive or a static string for copyright information purposes, the compiler automatically places these strings in the static data area, and generates references to these static data areas in the object code.

When **implicitstaticref** is in effect, any references to the static area by functions that are removed by the linker's garbage collection procedures are maintained; this may result in unresolved function definition errors by the linker.

When **noimplicitstaticref** is in effect, these references to the static area are removed, allowing for successful linking and potentially reduced executable size; note, however, that this may result in a failure to include the static data area and any copyright information that it may contain.

Specifying **-qfuncsect** with no suboption implies **implicitstaticref**.

Usage

Using multiple control sections increases the size of the object file, but can reduce the size of the final executable by allowing the linker to remove functions that are not called or that have been inlined by the optimizer at all places they are called.

The pragma directive must be specified before the first statement in the compilation unit.

Predefined macros

None.

Related information

- “#pragma comment” on page 278
- “-qkeepinlines (C++ only)” on page 143

-g

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates debug information for use by a symbolic debugger.

Syntax

►► — -g ————— ►►

Defaults

Not applicable.

Usage

Specifying **-g** will turn off all inlining unless you explicitly request it with an optimization option.

To specify that source files used with **-g** are referred to by either their absolute or their relative path name, use the **-qfullpath** option.

You can also use the **-qlinedebug** option to produce abbreviated debugging information in a smaller object size.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce an executable program testing so you can debug it, enter:

```
invocation myprogram.c -o testing -g
```

Related information

- “-qfullpath” on page 106
- “-qlinedebug” on page 163
- “-O, -qoptimize” on page 179

-qgcc_c_stdinc (C only)

Category

Compiler customization

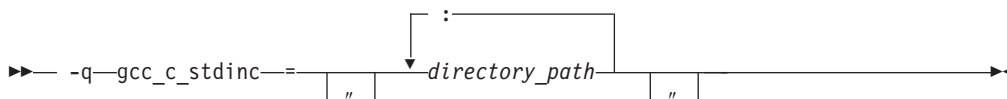
Pragma equivalent

None.

Purpose

Changes the standard search location for the GNU C system header files.

Syntax



Defaults

By default, the compiler searches the directory specified in the configuration file.

Parameters

directory_path

The path for the directory where the compiler should search for the GNU C header files. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the GNU C headers, you use a configuration file to do so; see “Directory search sequence for include files” on page 12 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the **-qnostdinc** option is in effect.

Predefined macros

None.

Examples

To override the default search paths for the GNU C headers with mypath/headers1 and mypath/headers2, enter:

```
invocation myprogram.c -qgcc_c_stdinc=mypath/headers1:mypath:headers2
```

Related information

- “-qc_stdinc (C only)” on page 78
- “-qstdinc” on page 225
- “-qinclude” on page 122
- “Directory search sequence for include files” on page 12
- “Specifying compiler options in a configuration file” on page 8

-qgcc_cpp_stdinc (C++ only)

Category

Compiler customization

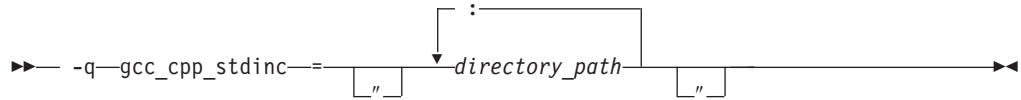
Pragma equivalent

None

Purpose

Changes the standard search location for the GNU C++ system header files.

Syntax



Defaults

By default, the compiler searches the directory specified in the configuration file.

Parameters

directory_path

The path for the directory where the compiler should search for the GNU C++ header files. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the GNU C++ headers, you use a configuration file to do so; see “Directory search sequence for include files” on page 12 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the **-qnostdinc** option is in effect.

Predefined macros

None.

Examples

To override the default search paths for the GNU C++ headers with mypath/headers1 and mypath/headers2, enter:

```
ppuxlcpp myprogram.C -qgcc_cpp_stdinc=mypath/headers1:mypath/headers2
```

Related information

- “-qcpp_stdinc (C++ only)” on page 79
- “-qstdinc” on page 225
- “-qinclude” on page 122
- “Directory search sequence for include files” on page 12
- “Specifying compiler options in a configuration file” on page 8

-qgenproto (C only)

Category

Portability and migration

Pragma equivalent

None.

Purpose

Produces prototype declarations from K&R function definitions or function definitions with empty parentheses, and displays them to standard output.

The compiler accepts and compiles K&R function definitions or definitions with a function declarator with empty parentheses; however, these function definitions are considered by the C standard to be obsolete (the compiler will diagnose them if you enable the **-qinfo=obs** option). When **-qgenproto** is in effect, the compiler generates the corresponding prototype declarations and displays them to standard output. You can use this option to help you identify obsolete function definitions and automatically obtain equivalent prototypes.

Syntax



Defaults

-qnogenproto

Parameters

parmnames

Parameter names are included in the prototype. If you do not specify this suboption, parameter names will not be included in the prototype.

Predefined macros

None.

Examples

Compiling with **-qgenproto** for the following function definitions:

```
int foo(a, b)    // K&R function
{
    int a, b;
}

int faa(int i) { } // prototyped function

main() { // missing void parameter
}
```

produces the following output on the display:

```
int foo(int, int);
int main(void);
```

Specifying **-qgenproto=parmnames** produces:

```
int foo(int a, int b);
int main(void);
```

-qhalt

Category

Error checking and debugging

Pragma equivalent

#pragma options halt

Purpose

Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity you specify.

Syntax

-qhalt syntax — C



-qhalt syntax — C++



Defaults

-qhalt=s

Parameters

i Specifies that compilation is to stop for all types of errors: warning, error and informational. Informational diagnostics (I) are of the lowest severity.

w Specifies that compilation is to stop for warnings (W) and all types of errors.

► **C** **e**

Specifies that compilation is to stop for errors (E), severe errors (S), and unrecoverable errors (U).

s ► **C** Specifies that compilation is to stop for severe errors (S) and unrecoverable errors (U). ► **C++** Specifies that compilation is to stop for severe errors (S).

Usage

When the compiler stops as a result of the **halt** option, the compiler return code is nonzero. For a list of return codes, see “Compiler return codes” on page 17.

When **-qhalt** is specified more than once, the lowest severity level is used.

Diagnostic messages may be controlled by the **-qflag** option.

You can also instruct the compiler to stop compilation based on the number of errors of a type of severity by using the **-qmaxerr** option, which overrides **-qhalt**.

 You can also use the **-qhaltormsg** option to stop compilation according to error message number.

Predefined macros

None.

Examples

To compile `myprogram.c` so that compilation stops if a warning or higher level message occurs, enter:

invocation `myprogram.c -qhalt=w`

Related information

- “-qhaltormsg (C++ only)”
- “-qflag” on page 97
- “-qmaxerr” on page 170

-qhaltormsg (C++ only)

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Stops compilation before producing any object, executable, or assembler source files if a specified error message is generated.

Syntax

►► -qhaltormsg=*message_identifier* ◀◀

Defaults

Not applicable.

Parameters

message_identifier

Represents a message identifier. The message identifier must be in the following format:

15dd-number

where:

dd Is the two-digit code representing the compiler component that produces the message. See “Compiler message format” on page 15 for descriptions of these.

number
Is the message number.

Usage

When the compiler stops as a result of the **-qhaltonmsg** option, the compiler return code is nonzero.

Predefined macros

None.

Related information

- “Compiler messages” on page 15

-qhot

Category

Optimization and tuning

Pragma equivalent

#pragma novector, #pragma nosimd

Purpose

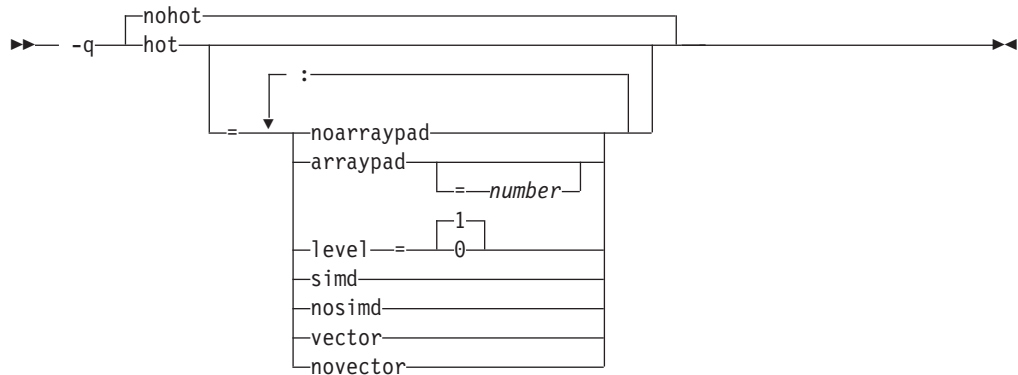
Performs high-order loop analysis and transformations (HOT) during optimization.

The **-qhot** compiler option is a powerful alternative to hand tuning that provides opportunities to optimize loops and array language. This compiler option will always attempt to optimize loops, regardless of the suboptions you specify.

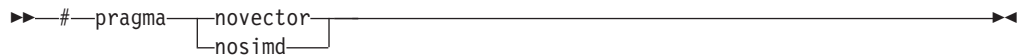
You can use the pragma directives to disable these transformations for selected sections of code.

Syntax

Option syntax



Pragma syntax



Defaults

- `-qnohot`
- `-qhot=noarraypad:level=0:nosimd:novector` when using **ppu** prefixed invocation commands and `-O3` is in effect.
- `-qhot=noarraypad:level=1:simd:vector` when using **ppu** prefixed invocation commands and `-O4` or `-O5` is in effect.
- `-qhot=noarraypad:level=1:simd:vector` when using **spu** prefixed invocation commands and `-O3` is in effect.
- Specifying `-qhot` without suboptions when using **ppu** prefixed invocation commands is equivalent to `-qhot=noarraypad:level=1:simd:vector`. Specifying `-qhot` without suboptions when using **spu** prefixed invocation commands is equivalent to `-qhot=noarraypad:level=1:simd:vector`.
- When compiling a program that targets both PPU and SPU using one of the **cbe** invocation commands, such as `cbexlc`, `-qsmp` is on by default and implies `-qhot=noarraypad:level=1:simd:vector`.

Parameters

arraypad | noarraypad (option only)

Permits the compiler to increase the dimensions of arrays where doing so might improve the efficiency of array-processing loops. (Because of the implementation of the cache architecture, array dimensions that are powers of two can lead to decreased cache utilization.) Specifying `-qhot=arraypad` when your source includes large arrays with dimensions that are powers of 2 can reduce cache misses and page faults that slow your array processing programs. This can be particularly effective when the first dimension is a power of 2. If you use this suboption with no *number*, the compiler will pad any arrays where it infers there may be a benefit and will pad by whatever amount it chooses. Not all arrays will necessarily be padded, and different arrays may be padded by different amounts. If you specify a *number*, the compiler will pad every array in the code.

Note: Using **arraypad** can be unsafe, as it does not perform any checking for reshaping or equivalences that may cause the code to break if padding takes place.

number (option only)

A positive integer value representing the number of elements by which each array will be padded in the source. The pad amount must be a positive integer value. It is recommended that pad values be multiples of the largest array element size, typically 4, 8, or 16.

level=0 (option only)

Performs a subset of the high-order transformations and sets the default to **novector:nosimd:noarraypad**.

level=1 (option only)

Performs the default set of high-order transformations.

simd (option only) | nosimd

When **simd** is in effect, the compiler converts certain operations that are performed in a loop on successive elements of an array into a call to a vector instruction. This call calculates several results at one time, which is faster than calculating each result sequentially. Applying this suboption is useful for applications with significant image processing demands.

nosimd disables the conversion of loop array operations into calls to vector instructions.

vector (option only) | novector (PPU only)

When specified with **-qnostrict** and **-qignerrno**, or an optimization level of **-O3** or higher, **vector** causes the compiler to convert certain operations that are performed in a loop on successive elements of an array (for example, square root, reciprocal square root) into a call to a routine in the Mathematical Acceleration Subsystem (MASS) library in libxlopt. If the operations are in a loop, the vector version of the routine is called. If the operations are scalar, the scalar version of the routine is called. The **vector** suboption supports single and double-precision floating-point mathematics, and is useful for applications with significant mathematical processing demands.

novector disables the conversion of loop array operations into calls to MASS library routines.

Since vectorization can affect the precision of your program's results, if you are using **-O4** or higher, you should specify **-qhot=novector** if the change in precision is unacceptable to you.

Usage

If you do not also specify an optimization level when specifying **-qhot** on the command line, the compiler assumes **-O2**.

When compiling a program that targets the PPU using one of the **ppu** invocation commands, such as **ppuxlc**, specifying **-O3** implies **-qhot=level=0**. When compiling a program that targets the SPU using one of the **spu** invocation commands, such as **spuxlc**, specifying **-O3** implies **-qhot=level=1**. To prevent all HOT optimizations with **-O3**, you must specify **-qnohot**.

If you want to override the default **level** setting of **1** when using **-O3** (for **spu** prefixed invocation commands only), **-O4** or **-O5**, be sure to specify **-qhot=level=0** after the other options.

The pragma directives apply only to while, do while, and for loops that immediately follow the placement of the directives. They have no effect on other loops that may be nested within the specified loop.

You can also use the **-qreport** option in conjunction with **-qhot** to produce a pseudo-C report showing how the loops were transformed; see “-qreport” on page 202 for details.

Predefined macros

None.

Examples

The following example shows the usage of **#pragma nosimd** to disable **-qhot=simd** for a specific for loop:

```
...
#pragma nosimd
for (i=1; i<1000; i++) {
    /* program code */
}
...
```

Related information

- “-qarch” on page 57
- “-qenablevmx (PPU only)” on page 94
- “-O, -qoptimize” on page 179
- “-qstrict” on page 227
- *Using the Mathematical Acceleration Subsystem (MASS) in the XL C/C++ Optimization and Programming Guide*

-I

Category

Input control

Pragma equivalent

None.

Purpose

Adds a directory to the search path for include files.

Syntax

►► — **-I**—*directory_path*—————►◄

Defaults

See “Directory search sequence for include files” on page 12 for a description of the default search paths.

Parameters

directory_path

The path for the directory where the compiler should search for the header files.

Usage

If **-qnostdinc** is in effect, the compiler searches *only* the paths specified by the **-I** option for header files, and not the standard search paths as well. If **-qidirfirst** is in effect, the directories specified by the **-I** option are searched before any other directories.

If the **-I** directory option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first. The **-I** directory option can be specified more than once on the command line. If you specify more than one **-I** option, directories are searched in the order that they appear on the command line.

The **-I** option has no effect on files that are included using an absolute path name.

Predefined macros

None.

Examples

To compile `myprogram.c` and search `/usr/tmp` and then `/oldstuff/history` for included files, enter:

```
invocation myprogram.c -I/usr/tmp -I/oldstuff/history
```

Related information

- “-qidirfirst”
- “-qstdinc” on page 225
- “-qinclude” on page 122
- “Directory search sequence for include files” on page 12
- “Specifying compiler options in a configuration file” on page 8

-qidirfirst

Category

Input control

Pragma equivalent

```
#pragma options [no]idirfirst
```

Purpose

Specifies whether the compiler searches for user include files in directories specified by the **-I** option *before* or *after* searching any other directories.

When **-qidirfirst** is in effect, the compiler first searches the directories specified by the **-I** option before searching any other directories. When **-qnoidirfirst** is in effect, before searching directories named on the **-I** option, the compiler first searches a) the directories in which source files named on the **-qinclude** option are located;

and b) the directories in which the including files are located.

Syntax

►► -q noidirfirst
idirfirst ◀◀

Defaults

-qnoidirfirst

Usage

This option only affects files included with the `#include "file_name"` directive or the **-qinclude** option; **-qidirfirst** is independent of the **-qnostdinc** option and has no effect on the search order for XL C/C++ or system header files. (For the search order of header files, see “Directory search sequence for include files” on page 12.) This option also has no effect on files that are included using an absolute path name.

The last valid pragma directive remains in effect until replaced by a subsequent pragma.

Predefined macros

None.

Examples

To compile `myprogram.c` and search `/usr/tmp/myinclude` for included files before searching the current directory (where the source file resides), enter:

invocation `myprogram.c -I/usr/tmp/myinclude -qidirfirst`

Related information

- “-I” on page 118
- “-qinclude” on page 122
- “-qstdinc” on page 225
- “-qc_stdinc (C only)” on page 78
- “-qcpp_stdinc (C++ only)” on page 79
- “Directory search sequence for include files” on page 12

-qignerrno

Category

Optimization and tuning

Pragma equivalent

`#pragma options [no]ignerrno`

Purpose

Allows the compiler to perform optimizations that assume `errno` is not modified by system calls.

Some system library functions set `errno` when an exception occurs. When **`ignerrno`** is in effect, the setting and subsequent side effects of `errno` are ignored. This allows the compiler to perform optimizations that assume `errno` is not modified by system calls.

Syntax

►► — `-q` — `noignerrno` — `ignerrno` —►►

Defaults

- `-qnoignerrno`
- **`-qignerrno`** when **`-O3`** or higher optimization is in effect.

Usage

If you require both **`-O3`** or higher and the ability to set `errno`, you should specify **`-qnoignerrno`** *after* the optimization option on the command line.

Predefined macros

► C++ `__IGNERRNO__` is defined to 1 when **`ignerrno`** is in effect; otherwise, it is undefined.

Related information

- “`-O`, `-qoptimize`” on page 179

`-qignprag`

Category

Language element control

Pragma equivalent

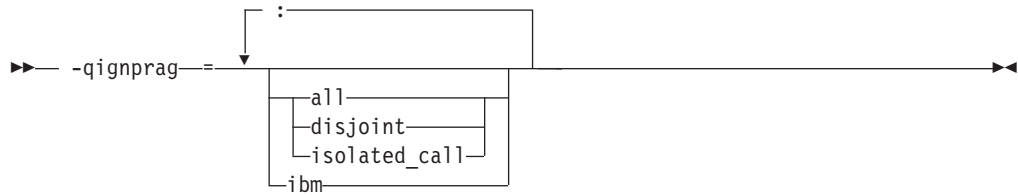
`#pragma options [no]ignprag`

Purpose

Instructs the compiler to ignore certain pragma statements.

This option is useful for detecting aliasing pragma errors. Incorrect aliasing gives runtime errors that are hard to diagnose. When a runtime error occurs, but the error disappears when you use **`ignprag`** with the **`-O`** option, the information specified in the aliasing pragmas is likely incorrect.

Syntax



Defaults

Not applicable.

Parameters

all

Ignores all **#pragma isolated_call** and **#pragma disjoint** directives in the source file.

disjoint

Ignores all **#pragma disjoint** directives in the source file.

ibm



Ignores all **#pragma ibm snapshot** directives in the source file.

isolated_call

Ignores all **#pragma isolated_call** directives in the source file.

Predefined macros

None.

Examples

To compile myprogram.c and ignore any **#pragma isolated_call** directives, enter:

```
invocation myprogram.c -qignprag=isolated_call
```

Related information

- “#pragma disjoint” on page 280
- “-qisolated_call” on page 141
- “#pragma ibm snapshot” on page 286

-qinclude

Category

Input control

Pragma equivalent

None.

Purpose

Specifies additional header files to be included in a compilation unit, as though the files were named in an **#include** statement in the source file.

The headers are inserted before all code statements and any headers specified by an `#include` preprocessor directive in the source file.

This option is provided for portability among supported platforms.

Syntax

► — `-qinclude=—file_path` —►

Defaults

Not applicable.

Parameters

file_path

The absolute or relative path and name of the header file to be included in the compilation units being compiled. If *file_path* is specified with a relative path, the search for it follows the sequence described in “Directory search sequence for include files” on page 12.

Usage

-qinclude is applied only to the files specified in the same compilation as that in which the option is specified. It is not passed to any compilations that occur during the link step, nor to any implicit compilations, such as those invoked by the option **-qtemplateregistry**, nor to the files generated by **-qtempinc**.

When the option is specified multiple times in an invocation, the header files are included in order of appearance on the command line. If the same header file is specified multiple times with this option, the header is treated as if included multiple times by `#include` directives in the source file, in order of appearance on the command line.

► **C++** When used with **-qtemplateregistry**, **-qinclude** is recorded in the template registry file, along with the source files affected by it. When these file dependencies initiate recompilation of the template registry, the **-qinclude** option is passed to the dependent files only if it had been specified for them when they were added to the template registry.

If you generate a listing file with **-qsource**, the header files included by **-qinclude** do not appear in the source section of the listing. Use **-qshowinc=usr** or **-qshowinc=all** in conjunction with **-qsource** if you want these header files to appear in the listing.

Any pragma directives that must appear before noncommentary statements in a source file will be affected; you cannot use **-qinclude** to include files if you need to preserve the placement of these pragmas.

Predefined macros

None.

Examples

To include the files `foo1.h` and `foo2.h` in the source file `foo.c`, enter:

```
invocation -qinclude=foo1.h foo.c -qinclude=foo2.h
```

Related information

- “Directory search sequence for include files” on page 12

-qinfo

Category

Error checking and debugging

Pragma equivalent

`#pragma options [no]info`, `#pragma info`

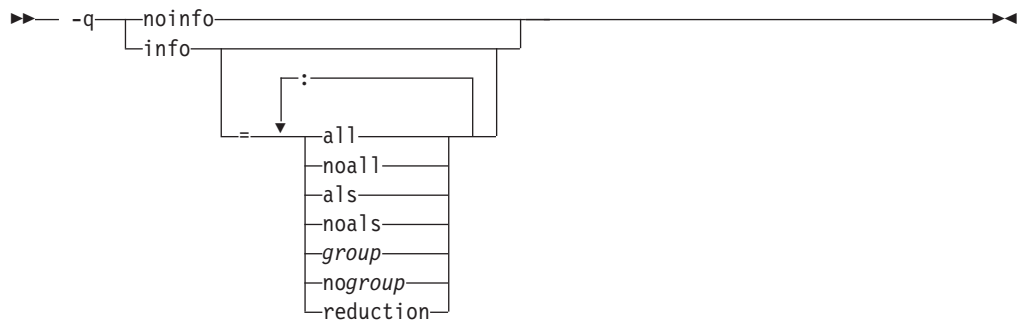
Purpose

Produces or suppresses groups of informational messages.

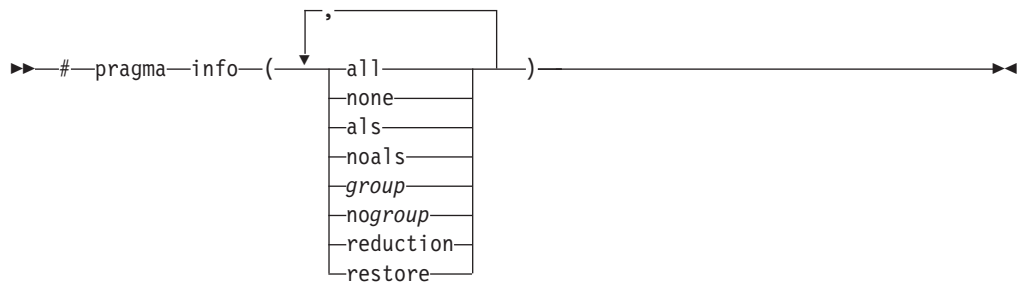
The messages are written to standard output and, optionally, to the listing file if one is generated.

Syntax



Option syntax



Pragma syntax



Defaults

-  -qnoinfo
-  -qinfo=lan:trx

Parameters

all Enables all diagnostic messages for all groups.

noall (option only)

Disables all diagnostic messages for all groups.

none (pragma only)

Disables all diagnostic messages for all groups.

als

Enables reporting possible violations of the ANSI aliasing rule in effect.

noals

Disables reporting possible aliasing-rule violations.

reduction

Lists all variables that are recognized as reduction variables inside a parallel loop.

group | **nogroup**

Enables or disables specific groups of messages, where *group* can be one or more of:

group Type of informational messages returned or suppressed

 **C**

c99 | **noc99**

C code that may behave differently between C89 and C99 language levels.

 **C++**

cls | **nocls**

C++ classes.

cmp | **nocmp**

Possible redundancies in unsigned comparisons.

cnd | **nocnd**

Possible redundancies or problems in conditional expressions.

cns | **nocns**

Operations involving constants.

cnv | **nocnv**

Conversions.

dcl | **nodcl**

Consistency of declarations.

eff | **noeff**

Statements and pragmas with no effect.

enu | **noenu**

Consistency of enum variables.

ext | **noext**

Unused external definitions.

gen | **nogen**

General diagnostic messages.

gnr | **nognr**
Generation of temporary variables.

got | **nogot**
Use of goto statements.

ini | **noini**
Possible problems with initialization.

lan | **nolan**
Language level effects.

obs | **noobs**
Obsolete features.

ord | **noord**
Unspecified order of evaluation.

par | **nopar**
Unused parameters.

por | **nopor**
Nonportable language constructs.

ppc | **noppc**
Possible problems with using the preprocessor.

ppt | **noppt**
Trace of preprocessor actions.

pro | **nopro**
Missing function prototypes.

rea | **norea**
Code that cannot be reached.

ret | **noret**
Consistency of return statements.

trd | **notrd**
Possible truncation or loss of data or precision.


tru | **notru**
Variable names truncated by the compiler.

trx | **notrx**
Hexadecimal floating point constants rounding.

uni | **nouni**
Uninitialized variables.

upg | **noupg**
Generates messages describing new behaviors of the current compiler release as compared to the previous release.

use | **nouse**
Unused auto and static variables.

 **vft** | **novft**
Generation of virtual function tables.

zea | **nozea**
Zero-extent arrays.

restore (pragma only)

Discards the current pragma setting and reverts to the setting specified by the previous pragma directive. If no previous pragma was specified, reverts to the command-line or default option setting.

► **C** Specifying **-qinfo** with no suboptions is equivalent to **-qinfo=all**.

► **C++** Specifying **-qinfo** with no suboptions is equivalent to **-qinfo=all:noppt**.

Usage

Specifying **-qnoinfo** is equivalent to **-qinfo=noall**.

Consider the following when enabling the reporting of aliasing-rule violations:

- **-qalias=ansi** must be set before reporting of aliasing-rule violations (**-qinfo=als**) can occur.
- Any level of optimization or inlining implies **-qinfo=noals** and a warning will be issued.
- Diagnostics are heuristic and may emit false positives. Points-to analysis cannot be evaluated deterministically in static compilation. The points-to analysis used for diagnostics is evaluated in a context-and-flow, insensitive manner. The sequence of traceback messages in diagnostics is such that if executed in the order specified, the indirect expression will point to the offending object. If that execution sequence cannot occur in the application, the diagnostic is a false positive. (See the **Examples** section for the types of diagnostics that can occur.)

Predefined macros

None.

Examples

To compile `myprogram.c` to produce informational message about all items except conversions and unreachable statements, enter:

invocation `myprogram.c -qinfo=all -qinfo=nocnv:norea`

► **C** The following example shows code constructs that the compiler detects when the code is compiled with **-qinfo=cnd:eff:got:obs:par:pro:rea:ret:uni** in effect:

```
#define COND 0

void faa() // Obsolete prototype (-qinfo=obs)
{
    printf("In faa\n"); // Unprototyped function call (-qinfo=pro)
}

int foo(int i, int k)
{
    int j; // Uninitialized variable (-qinfo=uni)

    switch(i) {
    case 0:
        i++;
        if (COND) // Condition is always false (-qinfo=cnd)
            i--; // Unreachable statement (-qinfo=rea)
        break;

    case 1:
```

```

        break;
        i++; // Unreachable statement (-qinfo=rea)
    default:
        k = (i) ? (j) ? j : i : 0;
}

    goto L; // Use of goto statement (-qinfo=got)
    return 3; // Unreachable statement (-qinfo=rea)
L:
    faa(); // faa() does not have a prototype (-qinfo=pro)

// End of the function may be reached without returning a value
// because of there may be a jump to label L (-qinfo=ret)

} //Parameter k is never referenced (-qinfo=ref)

int main(void) {
    ({ int i = 0; i = i + 1; i; }); // Statement does not have side effects (-qinfo=eff)

    return foo(1,2);
}

```

► C++ The following example shows code constructs that the compiler detects, with this code is compiled with **-qinfo=cls:cnd:eff:use** in effect:

```

#pragma abc // pragma not supported (-qinfo=eff or -qinfo=gen)

int bar() __attribute__((xyz)); // attribute not supported (-qinfo=eff)
int j();

class A {
    public:
        A(): x(0), y(0), z(0) { }; // this constructor is in the correct order
                                   // hence, no info message.
        A(int m): y(0), z(0)
        { x=m; }; // suggest using member initialization list
                  for x (-qinfo=cls)

        A(int m, int n):
        x(0), z(0) { }; // not all data members are initialized
                       // namely, y is not initialized (-qinfo=cls)

        A(int m, int n, int* l):
        x(m), z(l), y(n) { }; // order of class initialization (-qinfo=cls)

    private:
        int x;
        int y;
        int *z; // suggest having user-defined copy constructor/
                // assignment operator to handle the pointer data member
                // (-qinfo=cls)
};

int foo() {
    int j=5;
    j; // null statement (-qinfo=eff)
       // The user may mean to call j().

    return j;
}

void boo() {
    int x;
    int *i = &x;
    float *f; // f is not used (-qinfo=use)
    f = (float *) i; // incompatible type (-qinfo=eff)
}

```

```

// With ansi aliasing mode, a float pointer
// is not supposed to point to an int
}

void cond(int y) {
    const int i=0;
    int j;
    int k=0;

    if (i) {          // condition is always false (-qinfo=cnd)
        j=3;
    }

    if (1) {          // condition is always true (-qinfo=cnd)
        j=4;
    }

    j=0;
    if (j==0) {       // cond. is always true (-qinfo=cnd)
        j=5;
    }

    if (y) {
        k+=5
    }

    if (k==5) {       // This case cannot be determined, because k+=5
        // is in a conditional block.
        j=6;
    }
}

```

In the following example, the **#pragma info(eff, nouni)** directive preceding MyFunction1 instructs the compiler to generate messages identifying statements or pragmas with no effect, and to suppress messages identifying uninitialized variables. The **#pragma info(restore)** directive preceding MyFunction2 instructs the compiler to restore the message options that were in effect before the **#pragma info(eff, nouni)** directive was specified.

```

#pragma info(eff, nouni)
int MyFunction1()
{
    .
    .
    .
}

#pragma info(restore)
int MyFunction2()
{
    .
    .
    .
}

```

The following example shows a valid diagnostic for an aliasing violation:

```

t1.c:
int main() {
    short s = 42;
    int *pi = (int*) &s;
    *pi = 63;
    return 0;
}
x1C -+ -qinfo=als t1.c

```

"t1.c", line 4.3: 1540-0590 (I) Dereference may not conform to the current aliasing rules.
 "t1.c", line 4.3: 1540-0591 (I) The dereferenced expression has type "int". "pi" may point to "s" which has incompatible type "short".
 "t1.c", line 4.3: 1540-0592 (I) Check assignment at line 3 column 11 of t1.c.

In the following example, the analysis is context insensitive in that the two calls to floatToInt are not distinguished. There is no aliasing violation in this example, but a diagnostic is still issued.

```
t2.c:
int* floatToInt(float *pf) { return (int*)pf; }
```

```
int main() {
    int i;
    float f;
    int* pi = floatToInt((float*)&i);
    floatToInt(&f);
    return *pi;
}
```

```
xlc -+ -qinfo=als t2.c
```

"t2.c", line 8.10: 1540-0590 (I) Dereference may not conform to the current aliasing rules.
 "t2.c", line 8.10: 1540-0591 (I) The dereferenced expression has type "int". "pi" may point to "f" which has incompatible type "float".
 "t2.c", line 8.10: 1540-0592 (I) Check assignment at line 7 column 14 of t2.c.
 "t2.c", line 8.10: 1540-0592 (I) Check assignment at line 1 column 37 of t2.c.
 "t2.c", line 8.10: 1540-0592 (I) Check assignment at line 6 column 11 of t2.c.

```
t3.c:
int main() {
    float f;
    int i = 42;
    int *p = (int*) &f;
    p = &i;
    return *p;
}
```

```
xlc -+ -qinfo=als t3.c
```

"t3.c", line 6.10: 1540-0590 (I) Dereference may not conform to the current aliasing rules.
 "t3.c", line 6.10: 1540-0591 (I) The dereferenced expression has type "int". "p" may point to "f" which has incompatible type "float".
 "t3.c", line 6.10: 1540-0592 (I) Check assignment at line 4 column 10 of t3.c.

Related information

- “-qflag” on page 97

-qinitauto

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]initauto

Purpose

Initializes uninitialized automatic variables to a specific value, for debugging purposes.

Syntax

►► `-q` `noinitauto`
`initauto` `==` `hex_value` ►►

Defaults

`-qnoinitauto`

Parameters

hex_value

A two-digit hexadecimal byte value.

Usage

This option generates extra code to initialize the value of automatic variables. It reduces the runtime performance of the program and should only be used for debugging.

Predefined macros

- C++ `__INITAUTO__` is defined to the hex value specified on the `-qinitauto` option or pragma; otherwise, it is undefined.
- C++ `__INITAUTO_W__` is defined to the hex value, repeated 4 times, specified on the `-qinitauto` option or pragma; otherwise, it is undefined.

Examples

To compile `myprogram.c` so that automatic variables are initialized to hex value FF (decimal 255), enter:

invocation `myprogram.c -qinitauto=FF`

-qinlglue (PPU only)

Category

Object code control

Pragma equivalent

`#pragma options [no]inlglue`

Purpose

When used with `-O2` or higher optimization, inlines glue code that optimizes external function calls in your application.

Glue code, generated by the linker, is used for passing control between two external functions. When `-qinlglue` is in effect, the optimizer inlines glue code for better performance. When `-qnoinlglue` is in effect, inlining of glue code is prevented.

Note:

This option is ignored as glue code is always generated.

Syntax

►► — -q noinline
inline —►►

Defaults

-qnoinline

Usage

Inlining glue code can cause the code size to grow. Specifying **-qcompact** overrides the **-qinline** setting to prevent code growth. If you want **-qinline** to be enabled, do not specify **-qcompact**.

Specifying **-qnoinline** or **-qcompact** can degrade performance; use these options with discretion.

The **-qinline** option only affects function calls through pointers or calls to an external compilation unit. For calls to an external function, you should specify that the function is imported by using, for example, the **-qprocimported** option.

Predefined macros

None.

Related information

- “-qcompact” on page 74
- “-qprocimported, -qprocllocal, -qprocunknown (PPU only)” on page 195
- “-qtune” on page 250

-qinline

See “-Q, -qinline ” on page 198.

-qipa

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables or customizes a class of optimizations known as interprocedural analysis (IPA).

IPA is a two-step process: the first step, which takes place during compilation, consists of performing an initial analysis and storing interprocedural analysis information in the object file. The second step, which takes place during linking, and causes a complete recompilation of the entire application, applies the optimizations to the entire program.

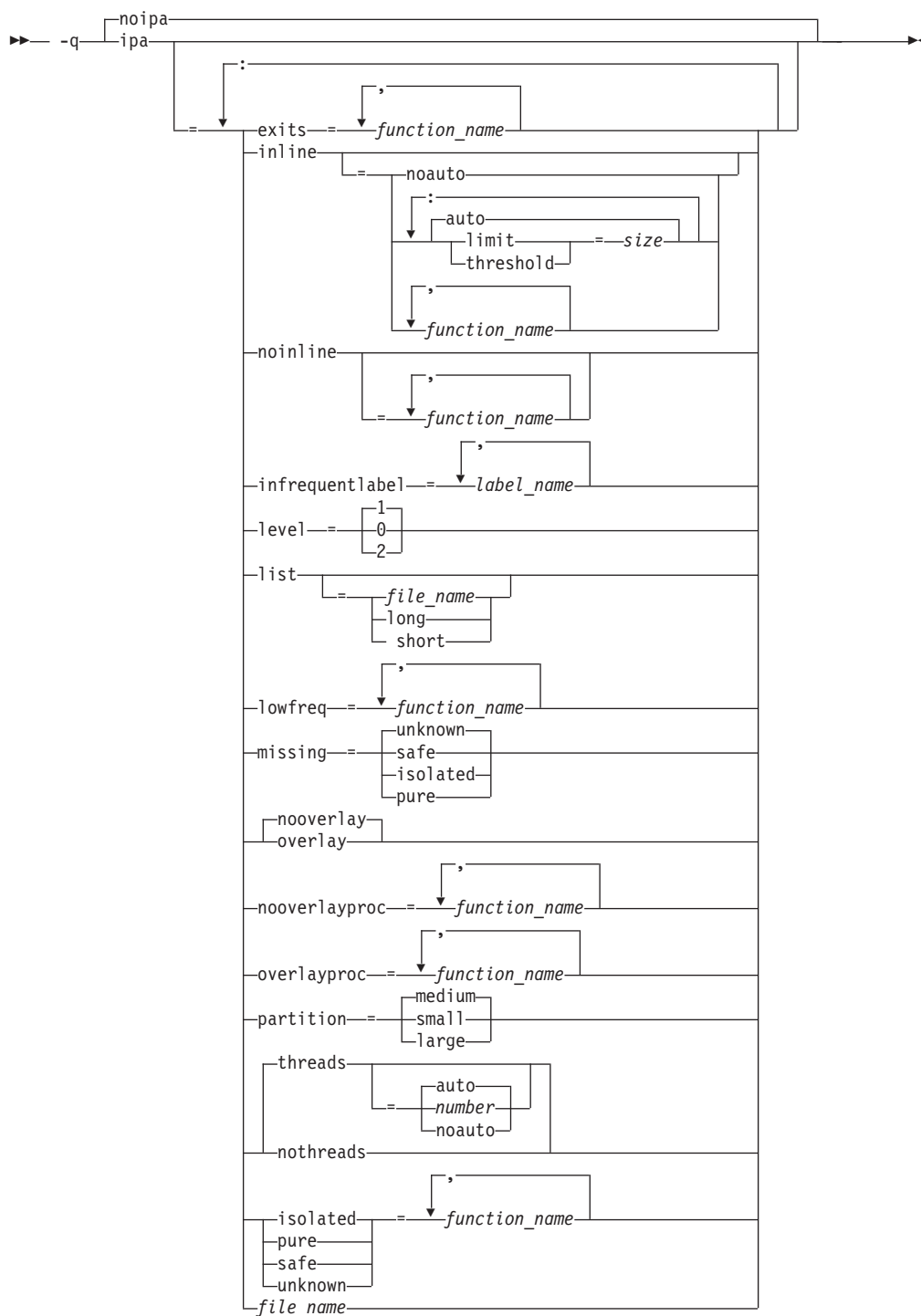
You can use **-qipa** during the compilation step, the link step, or both. If you compile and link in a single compiler invocation, only the link-time suboptions are relevant. If you compile and link in separate compiler invocations, only the compile-time suboptions are relevant during the compile step, and only the link-time suboptions are relevant during the link step.

Syntax

-qipa compile-time syntax



-qipa link-time syntax



Defaults

- **-qnoipa**
- **-qipa=inline=auto:level=1:missing=unknown:partition=medium:threads=auto**
for the PPU and
- **-qipa=inline=auto:level=1:missing=unknown:nooverlay:partition=medium:threads=auto**
for the SPU when **-O4** is in effect

- **-qipa=inline=auto:level=2:missing=unknown:partition=medium:threads=auto** for the PPU and
-qipa=inline=auto:level=2:missing=unknown:nooverlay:partition=medium:threads=auto for the SPU when **-O5** is in effect
- **-qipa=inline=auto:level=0:missing=unknown:partition=medium:threads=auto** for the PPU when **-qpdf1** or **-qpdf2** is in effect

Parameters

The following are parameters that may be specified during a separate compile step only:

object | **noobject**

Specifies whether to include standard object code in the output object files.

Specifying **noobject** can substantially reduce overall compile time by not generating object code during the first IPA phase. Note that if you specify **-S** with **noobject**, **noobject** will be ignored.


If compiling and linking are performed in the same step and you do not specify the **-S** or any listing option, **-qipa=noobject** is implied.

Specifying **-qipa** with no suboptions on the compile step is equivalent to **-qipa=object**.

The following are parameters that may be specified during a combined compile and link in the same compiler invocation, or during a separate link step only:

function_name

For all suboptions, the name of a function, or a comma-separated list of functions.

 Names must be specified using their mangled names. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file. (See also "Name mangling" in the *XL C/C++ Language Reference* for details on using the extern "C" linkage specifier on declarations to prevent name mangling.)

Regular expression syntax can be used to specify names for all suboptions that take function names as suboptions. Syntax rules for specifying regular expressions are described below:

Expression	Description
<i>string</i>	Matches any of the characters specified in <i>string</i> . For example, test will match testimony, latest, and intestine.
^ <i>string</i>	Matches the pattern specified by <i>string</i> only if it occurs at the beginning of a line.
<i>string</i> \$	Matches the pattern specified by <i>string</i> only if it occurs at the end of a line.
<i>str.i.ng</i>	The period (.) matches any single character. For example, t.st will match test, tast, tZst, and t1st.

Expression	Description
<i>string</i> \special_char	The backslash (\) can be used to escape special characters. For example, assume that you want to find lines ending with a period. Simply specifying the expression <i>.\$</i> would show all lines that had at least one character of any kind in it. Specifying <i>\.\$</i> escapes the period (.), and treats it as an ordinary character for matching purposes.
[<i>string</i>]	Matches any of the characters specified in <i>string</i> . For example, <i>t[a-g123]st</i> matches <i>tast</i> and <i>test</i> , but not <i>t-st</i> or <i>tAst</i> .
[^ <i>string</i>]	Does not match any of the characters specified in <i>string</i> . For example, <i>t[^a-zA-Z]st</i> matches <i>t1st</i> , <i>t-st</i> , and <i>t,st</i> but not <i>test</i> or <i>tYst</i> .
<i>string</i> *	Matches zero or more occurrences of the pattern specified by <i>string</i> . For example, <i>te*st</i> will match <i>tst</i> , <i>test</i> , and <i>teeeeeest</i> .
<i>string</i> +	Matches one or more occurrences of the pattern specified by <i>string</i> . For example, <i>t(es)+t</i> matches <i>test</i> , <i>tesest</i> , but not <i>tt</i> .
<i>string</i> ?	Matches zero or one occurrences of the pattern specified by <i>string</i> . For example, <i>te?st</i> matches either <i>tst</i> or <i>test</i> .
<i>string</i> { <i>m,n</i> }	Matches between <i>m</i> and <i>n</i> occurrence(s) of the pattern specified by <i>string</i> . For example, <i>a{2}</i> matches <i>aa</i> , and <i>b{1,4}</i> matches <i>b</i> , <i>bb</i> , <i>bbb</i> , and <i>bbbb</i> .
<i>string1</i> <i>string2</i>	Matches the pattern specified by either <i>string1</i> or <i>string2</i> . For example, <i>s o</i> matches both characters <i>s</i> and <i>o</i> .

exits

Specifies names of functions which represent program exits. Program exits are calls which can never return and can never call any function which has been compiled with IPA pass 1. The compiler can optimize calls to these functions (for example, by eliminating save/restore sequences), because the calls never return to the program. These functions must not call any other parts of the program that are compiled with **-qipa**.

infrequentlabel

Specifies user-defined labels that are likely to be called infrequently during a program run.

label_name

The name of a label, or a comma-separated list of labels.

inline

Enables function inlining by the high-level optimizer. Valid suboptions are any of the following:

auto | **noauto**

Enables or disables automatic function inlining by the high-level optimizer. When **-qipa=inline=auto** is in effect, the compiler considers all functions that are under the maximum size limit (see below) for inlining. When **-qipa=inline=noauto** is in effect, only functions listed in the *function_name* suboption are considered for inlining.

limit

When **-qipa=inline=auto** is in effect, specifies a limit on the size of a calling function after inlining.

threshold

When **-qipa=inline=auto** is in effect, specifies a limit on the size of a called function for it to be considered for inlining.

size

A nonnegative integer representing the relative size of function before and after inlining. The *size* is an arbitrary value representing a combination of factors, including the estimated size of the called function, the number of calls to the function, and so on. If you do not specify a *size*, the default is 1024 for the **threshold** suboption and 8192 for the **limit** suboption. Larger values for this number allow the compiler to inline larger functions, more function calls, or both.

Specifying **-qipa=inline** with no suboptions is equivalent to **-qipa=inline=auto**.

Note: By default, the compiler will try to inline all functions, not just those that you specified with the *function_name* suboption. If you want to turn on inlining for only certain functions, specify **inline=noauto** after you specify **inline=function_name**. (You must specify the suboptions in this order.) For example, to turn off inlining for all functions other than for *sub1*, specify **-qipa=inline=sub1:inline=noauto**.

noinline

When specified with no suboption, disables automatic function inlining by the high-level optimizer (equivalent to **-qipa=inline=noauto**). (Inlining may still be performed by the compiler front end or by the low-level optimizer; see “-Q, -qinline ” on page 198 for details.) When used with the *function_name* suboption, specifies functions that are not to be considered for automatic inlining by the high-level optimizer.

isolated

Specifies a comma-separated list of functions that are not compiled with **-qipa**. Functions that you specify as *isolated* or functions within their call chains cannot refer directly to any global variable.

level

Specifies the optimization level for interprocedural analysis. Valid suboptions are one of the following:

- 0** Performs only minimal interprocedural analysis and optimization.
- 1** Enables inlining, limited alias analysis, and limited call-site tailoring.
- 2** Performs full interprocedural data flow and alias analysis.

If you do not specify a level, the default is 1.

list

Specifies that a listing file be generated during the link phase. The listing file contains information about transformations and analyses performed by IPA, as well as an optional object listing for each partition.

If you do not specify a *list_file_name*, the listing file name defaults to *a.lst*. If you specify **-qipa=list** together with any other option that generates a listing file, IPA generates an *a.lst* file that overwrites any existing *a.lst* file. If you have a source file named *a.c*, the IPA listing will overwrite the regular compiler listing *a.lst*. You can use the **-qipa=list=list_file_name** suboption to specify an alternative listing file name.

Additional suboptions are one of the following:

- short** Requests less information in the listing file. Generates the Object File Map, Source File Map and Global Symbols Map sections of the listing.
- long** Requests more information in the listing file. Generates all of the

sections generated by the **short** suboption, plus the Object Resolution Warnings, Object Reference Map, Inliner Report and Partition Map sections.

lowfreq

Specifies functions that are likely to be called infrequently. These are typically error handling, trace, or initialization functions. The compiler may be able to make other parts of the program run faster by doing less optimization for calls to these functions.

missing

Specifies the interprocedural behavior of functions that are not compiled with **-qipa** and are not explicitly named in an **unknown**, **safe**, **isolated**, or **pure** suboption.

Valid suboptions are one of the following:

safe Specifies that the missing functions do not indirectly call a visible (not missing) function either through direct call or through a function pointer.

isolated

Specifies that the missing functions do not directly reference global variables accessible to visible function. Functions bound from shared libraries are assumed to be *isolated*.

pure Specifies that the missing functions are *safe* and *isolated* and do not indirectly alter storage accessible to visible functions. *pure* functions also have no observable internal state.

unknown

Specifies that the missing functions are not known to be *safe*, *isolated*, or *pure*. This suboption greatly restricts the amount of interprocedural optimization for calls to missing functions.

The default is to assume **unknown**.

overlay | nooverlay (SPU only)

Specifying **overlay** instructs the compiler to automatically create code overlays. The default, **nooverlay** specifies that the compiler should not automatically create code overlays. Related suboptions that allow for greater control of overlays are **overlayproc**, **nooverlayproc**, and **partition**. In the context of overlays, **partition** can be used to control the size of the overlay buffer. See "Using automatic code overlays" in the *XL C/C++ Optimization and Programming Guide* for more information such as using overlays with custom linker scripts.

nooverlayproc (SPU only)

Specifies a comma-separated list of functions in SPU programs that should not be overlaid. Use this option for functions that you always want as resident in the local store. C++ function names must be mangled. By default, no functions are exempt from being overlaid. This suboption has no effect if **-qipa=overlay** is not specified.

overlayproc (SPU only)

Specifies a comma-separated list of functions in SPU programs that should be in the same overlay. Multiple **overlayproc** suboptions may be present to specify multiple overlay groups. If a function is listed in multiple groups, it will be placed in the last group. C++ function names must be mangled. By default, no functions are in explicit overlays. This suboption has no effect if **-qipa=overlay** is not specified.

partition

Specifies the size of each program partition created by IPA during pass 2. Valid suboptions are one of the following:

- **small**
- **medium**
- **large**

Larger partitions contain more functions, which result in better interprocedural analysis but require more storage to optimize. Reduce the partition size if compilation takes too long because of paging. With SPU programs, when using **overlay**, the **partition** suboption controls the size of the overlay buffer.

pure

Specifies *pure* functions that are not compiled with **-qipa**. Any function specified as *pure* must be *isolated* and *safe*, and must not alter the internal state nor have side-effects, defined as potentially altering any data visible to the caller.

safe

Specifies *safe* functions that are not compiled with **-qipa** and do not call any other part of the program. Safe functions can modify global variables, but may not call functions compiled with **-qipa**.

threads | nothreads

Runs portions of the IPA optimization process during pass 2 in parallel threads, which can speed up the compilation process on multi-processor systems. Valid suboptions for the **threads** suboption are as follows:

auto | noauto

When **auto** is in effect, the compiler selects a number of threads heuristically based on machine load. When **noauto** is in effect, the compiler spawns one thread per machine processor.

number

Instructs the compiler to use a specific number of threads. *number* can be any integer value in the range of 1 to 32 767. However, *number* is effectively limited to the number of processors available on your system.

Specifying **threads** with no suboptions implies **-qipa=threads=auto**.

unknown

Specifies *unknown* functions that are not compiled with **-qipa**. Any function specified as *unknown* can make calls to other parts of the program compiled with **-qipa**, and modify global variables.

file_name

Gives the name of a file which contains suboption information in a special format.

The file format is the following:

```
# ... comment
attribute{, attribute} = name{, name}

missing = attribute{, attribute}
exits = name{, name}
lowfreq = name{, name}
inline
inline [ = auto | = noauto ]
inline = name{, name} [ from name{, name}]
inline-threshold = unsigned_int
inline-limit = unsigned_int
```

```
list [ = file-name | short | long ]
noinline
noinline = name{, name} [ from name{, name}]
level = 0 | 1 | 2
partition = small | medium | large
```

where *attribute* is one of:

- exits
- lowfreq
- unknown
- safe
- isolated
- pure

Specifying **-qipa** with no suboptions on the link step is equivalent to **-qipa=inline=auto:level=1:missing=unknown:partition=medium:threads=auto**.

Note: As of the V9.0 release of the compiler, the **pdfname** suboption is deprecated; you should use **-qpdf1=pdfname** or **-qpdf2=pdfname** in your new applications. See “-qpdf1, -qpdf2 (PPU only)” on page 187 for details.

Usage

Specifying **-qipa** automatically sets the optimization level to **-O2**. For additional performance benefits, you can also specify the **-Q** option. The **-qipa** option extends the area that is examined during optimization and inlining from a single function to multiple functions (possibly in different source files) and the linkage between them.

If any object file used in linking with **-qipa** was created with the **-qipa=noobject** option, any file containing an entry point (the main program for an executable program, or an exported function for a library) must be compiled with **-qipa**.

Some symbols which are clearly referenced or set in the source code may be optimized away by IPA, and may be lost to **debug** or **nm** outputs. Using IPA together with the **-g** compiler will usually result in non-steppable output.

Note that if you specify **-qipa** with **-#**, the compiler does not display linker information subsequent to the IPA link step.

For recommended procedures for using **-qipa**, see “Optimizing your applications” in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Examples

The following example shows how you might compile a set of files with interprocedural analysis:

```
invocation -c *.c -qipa
invocation -o product *.o -qipa
```

Here is how you might compile the same set of files, improving the optimization of the second compilation, and the speed of the first compile step. Assume that

there exist a set of routines, `user_trace1`, `user_trace2`, and `user_trace3`, which are rarely executed, and the routine `user_abort` that exits the program:

```
xlc -c *.c -qipa=noobject
xlc -c *.o -qipa=lowfreq=user_trace[123]:exit=user_abort
```

Related information

- “-Q, -qinline ” on page 198
- “-qisolated_call”
- “#pragma execution_frequency” on page 282
- “-qpdf1, -qpdf2 (PPU only)” on page 187
- “-S” on page 208
- “Optimizing your applications” in the *XL C/C++ Optimization and Programming Guide*
- “Using automatic code overlays” in the *XL C/C++ Optimization and Programming Guide*

-qisolated_call

Category

Optimization and tuning

Pragma equivalent

#pragma options isolated_call, #pragma isolated_call

Purpose

Specifies functions in the source file that have no side effects other than those implied by their parameters.

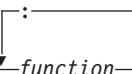
Essentially, any change in the state of the runtime environment is considered a side effect, including:

- Accessing a volatile object
- Modifying an external object
- Modifying a static object
- Modifying a file
- Accessing a file that is modified by another process or thread
- Allocating a dynamic object, unless it is released before returning
- Releasing a dynamic object, unless it was allocated during the same invocation
- Changing system state, such as rounding mode or exception handling
- Calling a function that does any of the above


Marking a function as isolated indicates to the optimizer that external and static variables cannot be changed by the called function and that pessimistic references to storage can be deleted from the calling function where appropriate. Instructions can be reordered with more freedom, resulting in fewer pipeline delays and faster execution in the processor. Multiple calls to the same function with identical parameters can be combined, calls can be deleted if their results are not needed, and the order of calls can be changed.

Syntax

Option syntax

►► `-qisolated_call=`  `function` ◀◀

Pragma syntax


►► `#pragma isolated_call`  `(function)` ◀◀

Defaults

Not applicable.

Parameters

function

The name of a function that does not have side effects or does not rely on functions or processes that have side effects. *function* is a primary expression that can be an identifier, operator function, conversion function, or qualified name. An identifier must be of type function or a typedef of function. 

If the name refers to an overloaded function, all variants of that function are marked as isolated calls.

Usage

The only side effect that is allowed for a function named in the option or pragma is modifying the storage pointed to by any pointer arguments passed to the function, that is, calls by reference. The function is also permitted to examine non-volatile external objects and return a result that depends on the non-volatile state of the runtime environment. Do not specify a function that causes any other side effects; that calls itself; or that relies on local static storage. If a function is incorrectly identified as having no side effects, the program behavior might be unexpected or produce incorrect results.

The **#pragma options isolated_call** directive must be placed at the top of a source file, before any statements. The **#pragma isolated_call** directive can be placed at any point in the source file, before or after calls to the function named in the pragma.

The **-qignprag** compiler option causes aliasing pragmas to be ignored; you can use **-qignprag** to debug applications containing the **#pragma isolated_call** directive.

Predefined macros

None.

Examples

To compile `myprogram.c`, specifying that the functions `myfunction(int)` and `classfunction(double)` do not have side effects, enter:

invocation `myprogram.c -qisolated_call=myfunction:classfunction`

The following example shows you when to use the **#pragma isolated_call** directive (on the addmult function). It also shows you when not to use it (on the same and check functions):

```
#include <stdio.h>
#include <math.h>

int addmult(int op1, int op2);
#pragma isolated_call(addmult)

/* This function is a good candidate to be flagged as isolated as its */
/* result is constant with constant input and it has no side effects. */
int addmult(int op1, int op2) {
    int rslt;

    rslt = op1*op2 + op2;
    return rslt;
}

/* The function 'same' should not be flagged as isolated as its state */
/* (the static variable delta) can change when it is called. */
int same(double op1, double op2) {
    static double delta = 1.0;
    double temp;

    temp = (op1-op2)/op1;
    if (fabs(temp) < delta)
        return 1;
    else {
        delta = delta / 2;
        return 0;
    }
}

/* The function 'check' should not be flagged as isolated as it has a */
/* side effect of possibly emitting output. */
int check(int op1, int op2) {
    if (op1 < op2)
        return -1;
    if (op1 > op2)
        return 1;
    printf("Operands are the same.\n");
    return 0;
}
```

Related information

- “-qignprag” on page 121
- “The const function attribute” and “The pure function attribute” in the *XL C/C++ Language Reference*

-qkeepinlines (C++ only)

Category

Object code control

Pragma equivalent

None.

Purpose

Keeps or discards definitions for unreferenced extern inline functions.

When **-qnokeepinlines** is in effect, definitions of unreferenced external inline functions are discarded. When **-qkeepinlines** is in effect, definitions of unreferenced external inline functions are kept.

Syntax



Defaults

-qnokeepinlines

Usage

-qnokeepinlines reduces the size of the object files.

When using **-qkeepinlines**, also specify the **-qfuncsect=noimplicitstaticref** option. The linker will keep only one version of a symbol and this may likely reduce the size of the final binary.

Predefined macros

None.

Related information

- “-qstaticinline (C++ only)” on page 222
- “-qfuncsect” on page 107

-qkeepparam

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

When used with **-O2** or higher optimization, specifies whether function parameters are stored on the stack.

A function usually stores its incoming parameters on the stack at the entry point. However, when you compile code with optimization options enabled, the compiler may remove these parameters from the stack if it sees an optimizing advantage in doing so. When **-qkeepparam** is in effect, parameters are stored on the stack even when optimization is enabled. When **-qnokeepparam** is in effect, parameters are removed from the stack if this provides an optimization advantage.

Syntax

►► -q nokeepparm
keepparm _____ ►►

Defaults

-qnokeepparm

Usage

Specifying **-qkeepparm** that the values of incoming parameters are available to tools, such as debuggers, by preserving those values on the stack. However, this may negatively affect application performance.

Predefined macros

None.

Related information

- “-O, -qoptimize” on page 179

-qkeyword

Category

Language element control

Pragma equivalent

None

Purpose

Controls whether the specified name is treated as a keyword or as an identifier whenever it appears in your program source.

Syntax

►► -q keyword
nokeyword = *keyword_name* _____ ►►

Defaults

By default all the built-in keywords defined in the C and C++ language standards are reserved as keywords.

Usage

You cannot add keywords to the language with this option. However, you can use **-qnokeyword=keyword_name** to disable built-in keywords, and use **-qkeyword=keyword_name** to reinstate those keywords.

► C++ This option can be used with all C++ built-in keywords.

► C This option can also be used with the following C keywords:

- `asm`
- `inline`
- `restrict`
- `typeof`

Note: ► C `asm` is not a keyword when the `-qlanglvl` option is set to `stdc89` or `stdc99`.

Predefined macros

- ► C++ `__BOOL__` is defined to 1 by default; however, it is undefined when `-qnokeyword=bool` is in effect.
- ► C `__C99_INLINE` is defined to 1 when `-qkeyword=inline` is in effect.
- `__C99_RESTRICT` is defined to 1 when `-qkeyword=restrict` is in effect.
- ► C `__IBM_GCC_ASM` is defined to 1 when `-qkeyword=asm` is in effect. (In C++ it is defined by default.)
- `__IBM_TYPEOF__` is defined to 1 when `-qkeyword=typeof` is in effect.

Examples

► C++ You can reinstate `bool` with the following invocation:
`ppuxlc++ -qkeyword=bool`

► C You can reinstate `typeof` with the following invocation:
`ppuxlc -qkeyword=typeof`

Related information

- “`-qasm`” on page 58

-l

Category

Linking

Pragma equivalent

None.

Purpose

Searches for the specified library file, *libkey.so*, and then *libkey.a* for dynamic linking, or just for *libkey.a* for static linking.

Syntax

►► -l—key—►►

Defaults

The compiler default is to search only some of the compiler runtime libraries. The default configuration file specifies the default library names to search for with the **-l** compiler option, and the default search path for libraries with the **-L** compiler option.

The C and C++ runtime libraries are automatically added.

Parameters

key

The name of the library minus the `lib` characters.

Usage

You must also provide additional search path information for libraries not located in the default search path. The search path can be modified with the **-L** option.

The **-l** option is cumulative. Subsequent appearances of the **-l** option on the command line do not replace, but add to, the list of libraries specified by earlier occurrences of **-l**. Libraries are searched in the order in which they appear on the command line, so the order in which you specify libraries can affect symbol resolution in your application.

For more information, refer to the **ld** documentation for your operating system.

Predefined macros

None.

Examples

To compile `myprogram.c` and link it with library `mylibrary` (`libmylibrary.a`) found in the `/usr/mylibdir` directory, enter:

```
invocation myprogram.c -lmylibrary -L/usr/mylibdir
```

Related information

- “**-L**”
- “Specifying compiler options in a configuration file” on page 8

-L

Category

Linking

Pragma equivalent

None.

Purpose

At link time, searches the directory path for library files specified by the **-l** option.

Syntax

►► — `-L`—*directory_path* —►►

Defaults

The default is to search only the standard directories. See the compiler configuration file for the directories that are set by default.

Parameters

directory_path

The path for the directory which should be searched for library files.

Usage

If the `-Ldirectory` option is specified both in the configuration file and on the command line, search paths specified in the configuration file are the first to be searched at link time.

For more information, refer to the `ld` documentation for your operating system.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the directory `/usr/tmp/old` is searched for the library `libspfiles.a`, enter:

```
invocation myprogram.c -lspfiles -L/usr/tmp/old
```

Related information

- “`-l`” on page 146

-qlanglvl

Category

Language element control

Pragma equivalent

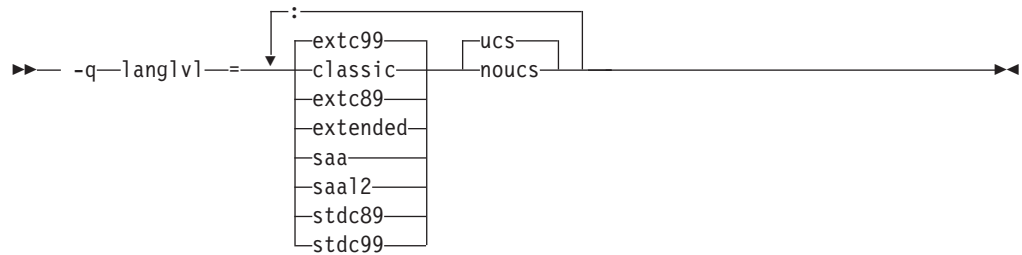
```
► C #pragma options langlvl, #pragma langlvl
```

Purpose

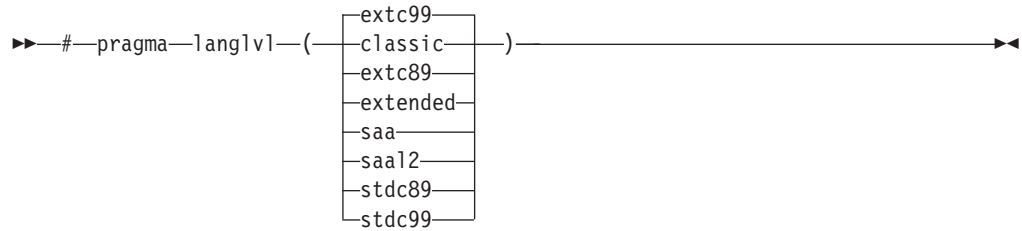
Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.

Syntax

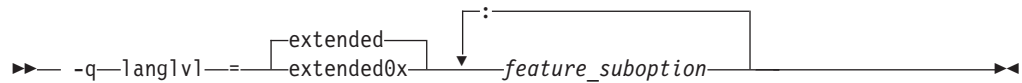
`-qlanglvl syntax` — **C**



#pragma langlvl syntax — C only



-qlanglvl syntax — C++



Defaults

- C
 The default is set according to the command used to invoke the compiler:
 - qlanglvl=extc99:ucs** for the **ppuxlc** or **spuxlc** and related invocation commands
 - qlanglvl=extended:noucs** for the **ppucc** or **spucc** and related invocation commands
 - qlanglvl=stdc89:noucs** for the **ppuc89** or **spuc89** and related invocation commands
 - qlanglvl=stdc99:ucs** for the **ppuc99** or **spuc99** and related invocation commands
- C++
-qlanglvl=extended:anonstruct:anonunion:ansifor:ansisinit:c99__func__:noc99complex:c99compo

Parameters

The following are the **-qlanglvl/#pragma langlvl** parameters for C language programs:

classic

Allows the compilation of nonstandard programs, and conforms closely to the K&R level preprocessor.

The following outlines the differences between the **classic** language level and all other standard-based language levels:

Tokenization

Tokens introduced by macro expansion may be combined with adjacent tokens in some cases. Historically, this was an artifact of the text-based implementations of older preprocessors, and because, in older implementations, the preprocessor was a separate program whose output was passed on to the compiler.

For similar reasons, tokens separated only by a comment may also be combined to form a single token. Here is a summary of how tokenization of a program compiled in **classic** mode is performed:

1. At a given point in the source file, the next token is the longest sequence of characters that can possibly form a token. For example, `i++++j` is tokenized as `i ++ ++ + j` even though `i ++ + ++ j` may have resulted in a correct program.
2. If the token formed is an identifier and a macro name, the macro is replaced by the text of the tokens specified on its `#define` directive. Each parameter is replaced by the text of the corresponding argument. Comments are removed from both the arguments and the macro text.
3. Scanning is resumed at the first step from the point at which the macro was replaced, as if it were part of the original program.
4. When the entire program has been preprocessed, the result is scanned again by the compiler as in the first step. The second and third steps do not apply here since there will be no macros to replace. Constructs generated by the first three steps that resemble preprocessing directives are not processed as such.

It is in the third and fourth steps that the text of adjacent but previously separate tokens may be combined to form new tokens.

The `\` character for line continuation is accepted only in string and character literals and on preprocessing directives.

Constructs such as:

```
#if 0
    "unterminated
#endif
#define US "Unterminating string
char *s = US terminated now"
```

will not generate diagnostic messages, since the first is an unterminated literal in a `FALSE` block, and the second is completed after macro expansion. However:

```
char *s = US;
```

will generate a diagnostic message since the string literal in `US` is not completed before the end of the line.

Empty character literals are allowed. The value of the literal is zero.

Preprocessing directives

The `#` token must appear in the first column of the line. The token immediately following `#` is available for macro expansion. The line can be continued with `\` only if the name of the directive and, in the following example, the `(` has been seen:

```
#define f(a,b) a+b
f\
(1,2)      /* accepted */
```

```
#define f(a,b) a+b
f(\
1,2)      /* not accepted */
```

The rules concerning \ apply whether or not the directive is valid. For example,

```
#\
define M 1  /* not allowed */

#def\
ine M 1     /* not allowed */

#define\
M 1         /* allowed */

#define\
M 1         /* equivalent to #define M 1, even
              though #define is not valid */
```

Following are the preprocessor directive differences.

#ifdef/#ifndef

When the first token is not an identifier, no diagnostic message is generated, and the condition is FALSE.

#else When there are extra tokens, no diagnostic message is generated.

#endif

When there are extra tokens, no diagnostic message is generated.

#include

The < and > are separate tokens. The header is formed by combining the spelling of the < and > with the tokens between them. Therefore /* and // are recognized as comments (and are always stripped), and the " and ' do begin literals within the < and >. (Remember that in C programs, C++-style comments // are recognized when **-qcplusplusmt** is specified.)

#line The spelling of all tokens which are not part of the line number form the new file name. These tokens need not be string literals.

#error

Not recognized.

#define

A valid macro parameter list consists of zero or more identifiers each separated by commas. The commas are ignored and the parameter list is constructed as if they were not specified. The parameter names need not be unique. If there is a conflict, the last name specified is recognized.

For an invalid parameter list, a warning is issued. If a macro name is redefined with a new definition, a warning will be issued and the new definition used.

#undef

When there are extra tokens, no diagnostic message is generated.

Macro expansion

- When the number of arguments on a macro invocation does not match the number of parameters, a warning is issued.
- If the (token is present after the macro name of a function-like macro, it is treated as too few arguments (as above) and a warning is issued.
- Parameters are replaced in string literals and character literals.

- Examples:

```
#define M()    1
#define N(a)   (a)
#define O(a,b) ((a) + (b))

M(); /* no error */
N(); /* empty argument */
O(); /* empty first argument
      and too few arguments */
```

Text output

No text is generated to replace comments.

extc89

Compilation conforms to the ANSI C89 standard, and accepts implementation-specific language extensions.

extc99

Compilation conforms to the ISO C99 standard, and accepts implementation-specific language extensions.

extended

Provides compatibility with the RT compiler and **classic**. This language level is based on C89.

saa

Compilation conforms to the current SAA[®] C CPI language definition. This is currently SAA C Level 2.

saal2

Compilation conforms to the SAA C Level 2 CPI language definition, with some exceptions.

stdc89

Compilation conforms strictly to the ANSI C89 standard, also known as ISO C90.

stdc99

Compilation conforms strictly to the ISO C99 standard.

ucs | noucs (option only)

Controls whether Unicode characters are allowed in identifiers, string literals and character literals in program source code. This suboption is enabled by default when **stdc99** or **extc99** is in effect. For details on the Unicode character set, see "The Unicode standard" in the *XL C/C++ Language Reference*.

The following **-qlanglvl** suboptions are accepted but ignored by the C compiler. Use **extended | extc99 | extc89** to enable the functions that these suboptions imply. For other language levels, the functions implied by these suboptions are disabled.

[no]gnu_assert

GNU C portability option.

[no]gnu_explicitregvar

GNU C portability option.

[no]gnu_include_next

GNU C portability option.

[no]gnu_locallabel

GNU C portability option.

[no]gnu_warning

GNU C portability option.

The following are the **-qlanglvl** parameters for C++ language programs:

extended

Compilation is based on the ISO C++ standard, with some differences to accommodate extended language features.

extended0x

A new group option has been created to compile code using all the features of the new version of the C++ standard (C++0x) that are implemented in IBM XL C/C++ for Multicore Acceleration for Linux, V10.1. Specifically, in this release we have introduced new integer promotion rules for arithmetic conversions with long long data types and we have enhanced the C++ preprocessor to support C99 features.

For more details about the new integer promotion rules, see "Integer promotion rules for C99 long long under C++" in the *XL C/C++ Language Reference*.

For more details about preprocessor changes, see "Preprocessor changes" in the *Getting Started with XL C/C++*.

Note: C++0x is a new version of the standard for the C++ programming language. This standard is temporary and has not been officially adopted in its entirety. Be aware that any future level of support for this standard is likely to change. Also see "Language levels and language extensions" in the *XL C/C++ Language Reference*.

feature_suboption

feature_suboption in the syntax diagram represents a colon-separated list of the remaining C++ options. They can be any of the following:

anonstruct | noanonstruct

Enables or disables support for anonymous structures and classes. Anonymous structures are typically used in unions, as in the following code fragment:

```
union U {
    struct {
        int i:16;
        int j:16;
    };
    int k;
} u;
// ...
u.j=3;
```

When the default, **-qlanglvl=anonstruct**, is in effect, anonymous structures are supported.

This is an extension to the C++ standard and gives behavior that is designed to be compatible with Microsoft® Visual C++. Specify **-qlanglvl=noanonstruct** for compliance with standard C++.

anonunion | noanonunion

Controls the members that are allowed in anonymous unions. When the default, **-qlanglvl=anonunion**, is in effect, anonymous unions can have members of all types that standard C++ allows in non-anonymous unions. For example, non-data members, such as structures, typedefs, and enumerations

are allowed. Member functions, virtual functions, or objects of classes that have non-trivial default constructors, copy constructors, or destructors cannot be members of a union, regardless of the setting of this option.

This is an extension to standard C++ and gives behavior that is designed to be compatible with Microsoft Visual C++. Specify **-qlanglvl=noanonunion** for compliance with standard C++.

ansifor | noansifor

Controls whether scope rules defined in the C++ standard apply to names declared in for loop initialization statements. When the default, **-qlanglvl=ansifor**, is in effect, standard C++ rules are used, and the following code causes a name lookup error:

```
{
    //...
    for (int i=1; i<5; i++) {
        cout << i * 2 << endl;
    }
    i = 10; // error
}
```

The reason for the error is that *i*, or any name declared within a for loop initialization statement, is visible only within the for statement. To correct the error, either declare *i* outside the loop or set **noansifor**.

When **-qlanglvl=noansifor** is in effect, the old language behavior is used; specify **-qlanglvl=noansifor** for compatibility with Microsoft Visual C++.

ansisinit | noansisinit

Controls whether standard C++ rules apply for handling static destructors for global and static objects. When the default, **-qlanglvl=ansisinit**, is in effect, the standard rules are used.

When **-qlanglvl=noansisinit** is in effect, the old language behavior is used.

c99_func | noc99_func

Enables or disables support for the C99 `__func__` identifier. For details of this feature, see "func_predefined identifier" in the *XL C/C++ Language Reference*.

c99complex | noc99complex

Enables or disables C99 complex data types and related keywords.

c99compoundliteral | noc99compoundliteral

Enables or disables support for C99 compound literals.

c99hexfloat | noc99hexfloat

Enables or disables support for C99-style hexadecimal floating constants.

c99vla | noc99vla

Enables or disables support for C99-type variable length arrays.

dependentbaselookup | nodependentbaselookup

Controls whether the name lookup rules for a template base class of dependent type defined in the Technical Corrigendum 1 (TC1) of the C++ Standard apply. Specify **-qlanglvl=nodependentbaselookup** for compliance with TC1. When **-qlanglvl=nodependentbaselookup** is in effect, unqualified names in a template class will not be resolved in a base class if that base class is dependent on a template parameter. These names must be qualified with the base class name in order to be found by name lookup. When the default, **-qlanglvl=dependentbaselookup**, is in effect, the behavior of previous XL C++ compilers remains.

The following example shows code that does not compile with **-q~~langlvl=~~nodependentbaselookup**:

```
struct base
{
    int baseName;
};

template <class B> struct derived : public B
{
    void func()
    {
        int i = baseName;    // this name will not be found in the base class
    };
};

int main(void)
{
    derived<base> x;
    x.func();
    return 0;
}
```

The following example shows code that compiles with or without **-q~~langlvl=~~nodependentbaselookup**:

```
struct base
{
    int baseName;
};

template <class B> struct derived : public B
{
    void func()
    {
        int i = B::baseName; // qualified name will be found in the base class
    };
};

int main(void)
{
    derived<base> x;
    x.func();
    return 0;
}
```

gnu_assert | nognu_assert

Enables or disables support for the following GNU C system identification assertions:

- **#assert**
- **#unassert**
- **#cpu**
- **#machine**
- **#system**

gnu_complex | nognu_complex

Enables or disables GNU complex data types and related keywords.

gnu_computedgoto | nognu_computedgoto

Enables or disables support for computed goto statements.

gnu_externtemplate | nognu_externtemplate

Enables or disables extern template instantiations. For details of this feature, see "Explicit instantiation" in the *XL C/C++ Language Reference*.

gnu_include_next | nognu_include_next

Enables or disables support for the GNU C #include_next preprocessor directive.

gnu_labelvalue | nognu_labelvalue

Enables or disables support for labels as values.

gnu_locallabel | nognu_locallabel

Enables or disables support for locally-declared labels.

gnu_membernamereuse | nognu_membernamereuse

Enables or disables reusing a template name in a member list as a typedef.

gnu_suffixij | nognu_suffixij

Enables or disables support for GNU-style complex numbers. When **-qlanglvl=gnu_suffixij** is in effect, a complex number can be ended with suffix i/I or j/J.

gnu_varargmacros | nognu_varargmacros

Enables or disables support for GNU-style macros with variable arguments. For details of this feature, see "Variadic macro extensions" in the *XL C/C++ Language Reference*.

gnu_warning | nognu_warning

Enables or disables support for the GNU C #warning preprocessor directive.

illptom | noillptom

Controls the expressions that can be used to form pointers to members. When the default, **-qlanglvl=illptom**, is in effect, the XL C++ compiler accepts some forms that are in common use but do not conform to the C++ Standard. For example, the following code defines a pointer to a function member, p, and initializes it to the address of C::foo, in the old style:

```
struct C {  
    void foo(int);  
};  
  
void (C::*p) (int) = C::foo;
```

This is an extension to standard C++ and gives behavior that is designed to be compatible with Microsoft Visual C++.

Specify **-qlanglvl=noillptom** for compliance with the C++ standard. The example code above must be modified to use the & operator.

```
struct C {  
    void foo(int);  
};  
  
void (C::*p) (int) = &C::foo;
```

implicitint | noimplicitint

Controls whether the compiler accepts missing or partially specified types as implicitly specifying int. When the default, **-qlanglvl=implicitint**, is in effect, a function declaration at namespace scope or in a member list will implicitly be declared to return int. Also, any declaration specifier sequence that does not completely specify a type will implicitly specify an integer type. The effect is as if the int specifier were present.

The following specifiers do not completely specify a type:

- auto
- const
- extern
- extern "literal"

- inline
- mutable
- friend
- register
- static
- typedef
- virtual
- volatile
- platform-specific types

For example, the return type of function `MyFunction` is `int` because it was omitted in the following code:

```
MyFunction()
{
    return 0;
}
```

Note that any situation where a type is specified is affected by this suboption. This includes, for example, template and parameter types, exception specifications, types in expressions (eg, casts, `dynamic_cast`, `new`), and types for conversion functions.

This is an extension to the C++ standard and gives behavior that is designed to be compatible with Microsoft Visual C++.

Specify **`-qlanglvl=noimplicitint`** for compliance with standard C++. For example, the function declaration above must be modified to:

```
int MyFunction()
{
    return 0;
}
```

`offsetnonpod` | `nooffsetnonpod`

Controls whether the `offsetof` macro can be applied to classes that are not data-only. C++ programmers often casually call data-only classes “Plain Old Data” (POD) classes. When the default, **`-qlanglvl=offsetnonpod`**, is in effect, you can apply `offsetof` to a class that contains one of the following:

- user-declared constructors or destructors
- user-declared assignment operators
- private or protected non-static data members
- base classes
- virtual functions
- non-static data members of type pointer to member
- a struct or union that has non-data members
- references

This is an extension to the C++ standard, and gives behavior that is designed to be compatible with Microsoft Visual C++. Specify

`-qlanglvl=nooffsetnonpod` for compliance with standard C++.

`olddigraph` | `noolddigraph`

Enables or disables support for old-style digraphs. When the default, **`-qlanglvl=olddigraph`**, is in effect, old-style digraphs are not supported. When **`-qlanglvl=olddigraph`** is in effect, the following digraphs are supported:

Digraph

Resulting character

%% # (pound sign)

%%%

(double pound sign, used as the preprocessor macro concatenation operator)

Specify **-qlanglvl=noolddigraph** for compatibility with standard C++.

This suboption only has effect when **-qdigraphs** is in effect.

oldfriend | nooldfriend

Controls whether friend declarations that name classes without elaborated class names are treated as C++ errors. When the default, **-qlanglvl=oldfriend**, is in effect, you can declare a friend class without elaborating the name of the class with the keyword `class`. For example, the statement below declares the class `IFont` to be a friend class:

```
friend IFont;
```

This is an extension to the C++ standard and gives behavior that is designed to be compatible with Microsoft Visual C++.

Specify the **-qlanglvl=nooldfriend** for compliance with standard C++. The example declaration above must be modified to the following:

```
friend class IFont;
```

oldtempacc | nooldtempacc

Controls whether access to a copy constructor to create a temporary object is always checked, even if creation of the temporary object is avoided. When the default, **-qlanglvl=oldtempacc**, is in effect, access checking is suppressed.

This is an extension to the C++ standard and gives behavior that is designed to be compatible with Microsoft Visual C++. Specify **-qlanglvl=nooldtempacc** for compliance with standard C++. For example, the `throw` statement in the following code causes an error because the copy constructor is a protected member of class `C`:

```
class C {
public:
    C(char *);
protected:
    C(const C&);
};

C foo() {return C("test");} // return copy of C object

void f()
{
    // catch and throw both make implicit copies of
    // the throw object
    throw C("error"); // throw a copy of a C object
    const C& r = foo(); // use the copy of a C object
    //                        created by foo()
}
```

The example code above contains three ill formed uses of the copy constructor `C(const C&)`.

oldtmplalign | nooldtmplalign

Controls whether alignment rules specified for nested templates are ignored.

When the default, **-qlanglvl=nooldtmplalign**, is in effect, these alignment rules are not ignored. For example, given the following template the size of `A<char>::B` will be 5 with **-qlanglvl=nooldtmplalign**, and 8 with **-qlanglvl=oldtmplalign** :

```
template <class T>
struct A {
#pragma options align=packed
    struct B {
        T m;
        int m2;
    };
#pragma options align=reset
};
```

oldtmpspec | nooldtmpspec

Controls whether template specializations that do not conform to the C++ standard are allowed. When the default, **-qlanglvl=oldtmpspec**, is in effect, you can explicitly specialize a template class as in the following example, which specializes the template class `ribbon` for type `char`:

```
template<class T> class ribbon { /*...*/};
class ribbon<char> { /*...*/};
```

This is an extension to standard C++ and gives behavior that is designed to be compatible with Microsoft Visual C++.

Specify **-qlanglvl=nooldtmpspec** for compliance with standard C++. In the example above, the template specialization must be modified to:

```
template<class T> class ribbon { /*...*/};
template<> class ribbon<char> { /*...*/};
```

redefmac | noredefmac

Controls whether a macro can be redefined without a prior `#undef` or `undefine()` statement.

trailenum | notrailenum

Controls whether trailing commas are allowed in enum declarations. When the default, **-qlanglvl=trailenum**, is in effect, one or more trailing commas are allowed at the end of the enumerator list. For example, the following enum declaration uses this extension:

```
enum grain { wheat, barley, rye,, };
```

This is an extension to the C++ standard, and is intended to provide compatibility with Microsoft Visual C++.

Specify **-qlanglvl=notrailenum** for compliance with standard C++.

typedefclass | notypedefclass

Controls whether a typedef name can be specified where a class name is expected. When the default, **-qlanglvl=typedefclass**, is in effect, the standard C++ rule applies, and a typedef name cannot be specified where a class name is expected. Specify **-qlanglvl=typedefclass** to allow the use of typedef names in base specifiers and constructor initializer lists.

ucs | noucs

Controls whether Unicode characters are allowed in identifiers, string literals and character literals in program source code. For details on the Unicode character set, see "The Unicode standard" in the *XL C/C++ Language Reference*.

varargmacros | novargmacros

Enables or disables support for C99-style variable argument lists in function-like macros. For details of this feature, see "Function-like macros" in the *XL C/C++ Language Reference*.

zeroextarray | nozeroextarray

Controls whether zero-extent arrays are allowed as the last non-static data


member in a class definition. When the default, **-qlanglvl=zeroextentarray**, is in effect, arrays with zero elements are allowed. The example declarations below define dimensionless arrays a and b.

```
struct S1 { char a[0]; };  
struct S2 { char b[]; };
```


This is an extension to the C++ standard, and is intended to provide compatibility with Microsoft Visual C++.

Specify **-qlanglvl=nozeroextarray** for compliance with standard C++.

Usage

 In general, if you specify a suboption with the **no** form of the option, the compiler will diagnose any uses of the feature in your code with a warning, unless you disable the warning with the **-qsuppress** option. Additionally, you can use the **-qinfo=por** option to generate informational messages along with the following suboptions:

- [no]c99complex
- [no]gnu_complex

 Since the pragma directive makes your code non-portable, it is recommended that you use the option rather than the pragma. If you do use the pragma, it must appear before any noncommentary lines in the source code. Also, because the directive can dynamically alter preprocessor behavior, compiling with the preprocessing-only options may produce results different from those produced during regular compilation.

Predefined macros

See “Macros related to language levels” on page 331 for a list of macros that are predefined by **-qlanglvl** suboptions.

Related information

- “-qsuppress” on page 231
- “The IBM XL C language extensions” in the *XL C/C++ Language Reference* and “The IBM XL C++ language extensions” in the *XL C/C++ Language Reference*

-qlargepage

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Takes advantage of large pages for applications designed to execute in a large page memory environment.

When **-qlargepage** is in effect to compile a program designed for a large page environment, an increase in performance can occur.

Syntax

►► `-q` no largepage
largepage ►►

Defaults

`-qnolargepage`

Usage

Note that this option is only useful in the following conditions:

- Large pages must be available and configured on the system.
- You must compile with an option that enables loop optimization, such as `-O3` or `-qhot`.
- You must link with the `-blpdata` option.

See your operating system documentation for more information on using large page support.

Predefined macros

None.

Examples

To compile `myprogram.c` to use large page heaps, enter:

invocation `myprogram.c -qlargepage -blpdata`

-qlib

Category

Linking

Pragma equivalent

None.

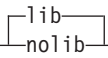
Purpose

Specifies whether standard system libraries and XL C/C++ libraries are to be linked.

When **-qlib** is in effect, the standard system libraries and compiler libraries are automatically linked. When **-qnolib** is in effect, the standard system libraries and compiler libraries are not used at link time; only the libraries specified on the command line with the `-l` flag will be linked.

This option can be used in system programming to disable the automatic linking of unneeded libraries.

Syntax

►► — -q —  —►►

Defaults

-qlib

Usage

Using **-qno lib** specifies that no libraries, including the system libraries as well as the XL C/C++ libraries (these are found in the `lib/` and `lib64/` subdirectories of the compiler installation directory), are to be linked. The system startup files are still linked, unless **-qnocrt** is also specified.

Note that if your program references any symbols that are defined in the standard libraries or compiler-specific libraries, link errors will occur. To avoid these unresolved references when compiling with **-qno lib**, be sure to explicitly link the required libraries by using the command flag **-l** and the library name.

Predefined macros

None.

Examples

To compile `myprogram.c` without linking to any libraries except the compiler library `libxlopt.a`, enter:

```
invocation myprogram.c -qno lib -lxlopt
```

Related information

- “-qcrt” on page 77

-qlibansi

Category

Optimization and tuning

Pragma equivalent

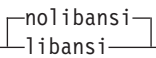
#pragma options [no]libansi

Purpose

Assumes that all functions with the name of an ANSI C library function are in fact the system functions.

When **libansi** is in effect, the optimizer can generate better code because it will know about the behavior of a given function, such as whether or not it has any side effects.

Syntax

►► — -q —  —►►

Defaults

-qnolibansi

Predefined macros

► C++ `__LIBANSI__` is defined to 1 when **libansi** is in effect; otherwise, it is not defined.

-qlinedebug

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates only line number and source file name information for a debugger.

When **-qlinedebug** is in effect, the compiler produces minimal debugging information, so the resulting object size is smaller than that produced by the **-g** debugging option. You can use the debugger to step through the source code, but you will not be able to see or query variable information. The traceback table, if generated, will include line numbers.

Syntax

►► — -q —  —►►

Defaults

-qnolinedebug

Usage

When **-qlinedebug** is in effect, function inlining is disabled.

Avoid using **-qlinedebug** with **-O** (optimization) option. The information produced may be incomplete or misleading.

The **-g** option overrides the **-qlinedebug** option. If you specify **-g** with **-qnolinedebug** on the command line, **-qnolinedebug** is ignored and a warning is issued.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce an executable program testing so you can step through it with a debugger, enter:

```
invocation myprogram.c -o testing -qlinedebug
```

Related information

- “-g” on page 108
- “-O, -qoptimize” on page 179

-qlist

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]list

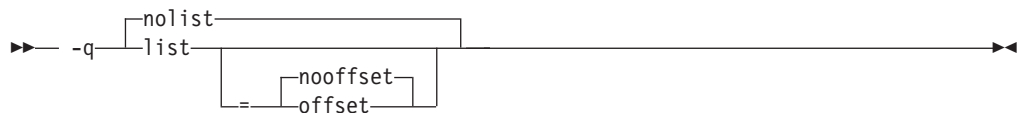
Purpose

Produces a compiler listing file that includes an object or assembly listing.

When **list** is in effect, a listing file is generated with a `.lst` suffix for each source file named on the command line. For details of the contents of the listing file, see “Compiler listings” on page 18.

You can use the object or assembly listing to help understand the performance characteristics of the generated code and to diagnose execution problems.

Syntax



Defaults

-qnolist

Parameters

offset | **nooffset** (PPU only)

Changes the offset of the PDEF header from 00000 to the offset of the start of the text area. Specifying the option allows any program reading the `.lst` file to add the value of the PDEF and the line in question, and come up with the same value whether **offset** or **nooffset** is specified. The **offset** suboption is only relevant if there are multiple procedures in a compilation unit.

Specifying **list** without the suboption is equivalent to **list=nooffset**.

Usage

The `-qnoprint` compiler option overrides this option.

Predefined macros

None.

Examples

To compile `myprogram.c` and to produce a listing (`.lst`) file that includes an object listing, enter:

```
invocation myprogram.c -qlist
```

Related information

- “`-qlistopt`”
- “`-qprint`” on page 193
- “`-qsource`” on page 218

-qlistopt

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Produces a compiler listing file that includes all options in effect at the time of compiler invocation.

When **listopt** is in effect, a listing file is generated with a `.lst` suffix for each source file named on the command line. The listing shows options in effect as set by the compiler defaults, the configuration file, and command line settings. For details of the contents of the listing file, see “Compiler listings” on page 18.

Syntax

```
►► — -q —┐ nolistopt
           └─ listopt —◄◄
```

Defaults

`-qnolistopt`

Usage

Option settings caused by pragma statements in the program source are not shown in the compiler listing.

The `-qnoprint` compiler option overrides this option.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce a listing (`.lst`) file that shows all options in effect, enter:

```
invocation myprogram.c -qlistopt
```

Related information

- “-qlist” on page 164
- “-qprint” on page 193
- “-qsource” on page 218

-qlonglit (PPU only)

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

In 64-bit mode, promotes literals with implicit type of `int` to `long`.

Syntax

►► -q nolonglit
longlit _____ ►►

Defaults

-qnolonglit

Usage

Note: You can only compile 64-bit code using one of the ppu invocation commands, such as **ppuxlc**. A program that targets both PPU and SPU and that is compiled using one of the **cbe** invocation commands, such as **cbexlc**, must be 32-bit. This option can not be used for such a program

The following table shows the default implicit types for constants and the implicit types when **-qlonglit** is in effect.

Suffix	Decimal literals		Hexadecimal or octal literals	
	Default implicit type	Implicit type with -qlonglit in effect	Default implicit type	Implicit type with -qlonglit in effect
unsuffixes	int long int	long int	int unsigned int long int unsigned long int	long int unsigned long int
u or U	unsigned int unsigned long int	unsigned long int	unsigned int unsigned long int	unsigned long int
l or L	long int	long int	long int unsigned long int	long int unsigned long int
Both u or U, and l or L	unsigned long int	unsigned long int	unsigned long int	unsigned long int
ll or LL	long long int	long long int	long long int unsigned long long int	long long int unsigned long long int
Both u or U, and ll or LL	unsigned long long int	unsigned long long int	unsigned long long int	unsigned long long int

Predefined macros

None.

-qlonglong Category

Language element control

Pragma equivalent

#pragma options [no]longlong

Purpose

Allows IBM long long integer types in your program.

Syntax

►► — -q — longlong — no longlong — ►►

Defaults

- **C** **-qlonglong** for the **ppucc** or **spucc** invocation command or the **-qlanglvl=extended** | **extc89** option; **-qno longlong** for the **ppuc89** or **spuc89** invocation command or **-qlanglvl=stdc89** option.
- **C++** **-qlonglong**

Usage

C This option only has an effect with the **ppucc** or **spucc** or **ppuc89** or **spuc89** invocation commands, or when the **-qlanglvl** option is set to **extended** | **stdc89** | **extc89**. It is not valid for the **ppuxlc** or **spuxlc** invocation command or when the language level **stdc99** | **extc99** is in effect, as the long long support provided by this option is incompatible with the semantics of the long long types mandated by the C99 standard.

C++ The functionality provided by this option is available with the **-qlanglvl=extended0x** option, as well. The only difference is that new unsigned integer promotion rules apply with that option. For more details about those rules see "Integer promotion rules for C99 long long under C++" in the *XL C/C++ Language Reference*.

Predefined macros

`_LONG_LONG` is defined to 1 when long long data types are available; otherwise, it is undefined.

Examples

To compile `myprogram.c` with support for IBM long long integers, enter:

```
ppucc myprogram.c -qlonglong
```

Related information

- "Integral types" in the *XL C/C++ Language Reference*

-ma (C only)

See "`-qalloca`, `-ma (C only)`" on page 55.

-qmakedep, -M

Category

Output control

Pragma equivalent

None.

Purpose

Creates an output file containing targets suitable for inclusion in a description file for the **make** command.

The output file is named with a `.d` suffix.

Syntax



Defaults

Not applicable.

Parameters

gcc (-qmakedep option only)

The format of the generated **make** rule to matches the GCC format: the description file includes a single target listing all of the main source file's dependencies.

If you specify **-qmakedep** with no suboption, or **-M**, the description file specifies a separate rule for each of the main source file's dependencies.

Usage

For each source file with a `.c`, `.C`, `.cpp`, or `.i` suffix named on the command line, an output file is generated with the same name as the object file and a `.d` suffix. Output files are not created for any other types of input files. If you use the **-o** option to rename the object file, the output file uses the name you specified on the **-o** option. See below for examples.

The output files generated by these options are not **make** files; they must be linked before they can be used with the **make** command. For more information on this command, see your operating system documentation.

The output file contains a line for the input file and an entry for each include file. It has the general form:

```
file_name.o:include_file_name
file_name.o:file_name.suffix
```

You can also use the following option with **qmakedep** and **-M**:

-MF=*file_path*

Sets the name of the output file, where *file_path* is the full or partial path or file name for the output file. See below for examples.

Include files are listed according to the search order rules for the `#include` preprocessor directive, described in "Directory search sequence for include files" on page 12. If the include file is not found, it is not added to the `.d` file.

Files with no include statements produce output files containing one line that lists only the input file name.

Predefined macros

None.

Examples

To compile `mysource.c` and create an output file named `mysource.d`, enter:

```
invocation -c -qmakedep mysource.c
```

To compile `foo_src.c` and create an output file named `mysource.d`, enter:

```
invocation -c -qmakedep foo_src.c -MF mysource.d
```

To compile `foo_src.c` and create an output file named `mysource.d` in the `deps/` directory, enter:

```
invocation -c -qmakedep foo_src.c -MF deps/mysource.d
```

To compile `foo_src.c` and create an object file named `foo_obj.o` and an output file named `foo_obj.d`, enter:

```
invocation -c -qmakedep foo_src.c -o foo_obj.o
```

To compile `foo_src.c` and create an object file named `foo_obj.o` and an output file named `mysource.d`, enter:

```
invocation -c -qmakedep foo_src.c -o foo_obj.o -MF mysource.d
```

To compile `foo_src1.c` and `foo_src2.c` to create two output files, named `foo_src1.d` and `foo_src2.d`, respectively, in the `c:/tmp/` directory, enter:

```
invocation -c -qmakedep foo_src1.c foo_src2.c -MF c:/tmp/
```

Related information

- “-MF” on page 174
- “-o” on page 178
- “Directory search sequence for include files” on page 12

-qmaxerr

Category

Error checking and debugging

Pragma equivalent

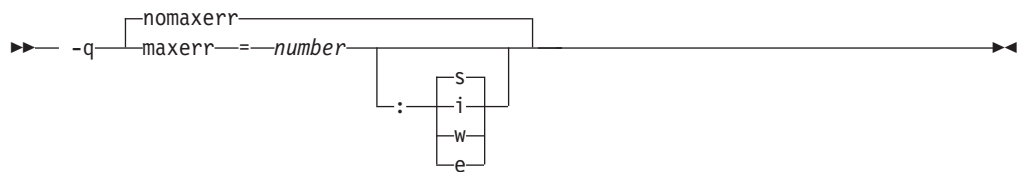
None.

Purpose

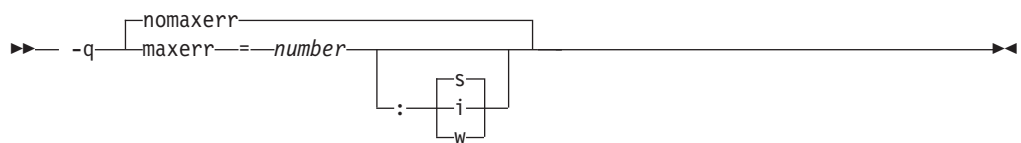
Halts compilation when a specified number of errors of a specified severity level or higher is reached.

Syntax

-qmaxerr syntax — C



-qmaxerr syntax — C++



Defaults

-qnomaxerr: The compiler continues to process as much input as possible, until it is not able to generate code.

Parameters

number

Must be an integer with a value of 1 or greater. An unrecoverable error occurs when the number of errors reaches the limit specified, and compilation stops.

i Specifies a minimum severity level of Informational (I).

w Specifies a minimum severity level of Warning (W).

c **e**

Specifies a minimum severity level of Error (E).

s Specifies a minimum severity level of Severe error (S).

If you specify **-qmaxerr** with no severity level and the **-qhalt** option or pragma is also in effect, the severity level specified by **halt** is used. If you specify **-qmaxerr** with no severity level and **halt** is not in effect, the default severity level is **s**.

Usage

If the **-qmaxerr** option is specified more than once, the **-qmaxerr** option specified last determines the action of the option. If both the **-qmaxerr** and **-qhalt** options are specified, the **-qmaxerr** or **-qhalt** option specified last determines the severity level used by the **-qmaxerr** option.

Diagnostic messages may be controlled by the **-qflag** option.

Predefined macros

None.

Examples

To stop compilation of `myprogram.c` when 10 warnings are encountered, enter the command:

```
invocation myprogram.c -qmaxerr=10:w
```

To stop compilation of `myprogram.c` when 5 severe errors are encountered, assuming that the current **-qhalt** option value is **s** (severe), enter the command:

```
invocation myprogram.c -qmaxerr=5
```

To stop compilation of `myprogram.c` when 3 informational messages are encountered, enter the command:

```
invocation myprogram.c -qmaxerr=3:i
```

or:

```
invocation myprogram.c -qmaxerr=3 -qhalt=i
```

Related information

- “**-qflag**” on page 97
- “**-qhalt**” on page 113
- “Message severity levels and compiler response” on page 16

-qmaxmem

Category

Optimization and tuning

Pragma equivalent

#pragma options maxmem

Purpose

Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.

Syntax

►► `-qmaxmem=size_limit` ◄◄

Defaults

- `-qmaxmem=8192` when `-O2` is in effect.
- `-qmaxmem=-1` when `-O3` or higher optimization is in effect.

Parameters

size_limit

The number of kilobytes worth of memory to be used by optimizations. The limit is the amount of memory for specific optimizations, and not for the compiler as a whole. Tables required during the entire compilation process are not affected by or included in this limit.

A value of -1 permits each optimization to take as much memory as it needs without checking for limits.

Usage

A smaller limit does not necessarily mean that the resulting program will be slower, only that the compiler may finish before finding all opportunities to increase performance. Increasing the limit does not necessarily mean that the resulting program will be faster, only that the compiler is better able to find opportunities to increase performance if they exist.

Setting a large limit has no negative effect on the compilation of source files when the compiler needs less memory. However, depending on the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, setting the limit too high, or to -1, might exceed available system resources.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the memory specified for local table is 16384 kilobytes, enter:

```
invocation myprogram.c -qmaxmem=16384
```

-qmbcs, -qdbcs

Category

Language element control

Pragma equivalent

```
#pragma options [no]mbcs, #pragma options [no]dbcs
```

Purpose

Enables support for multibyte character sets (MBCS) and Unicode characters in your source code.

When **mbcs** or **dbcs** is in effect, multibyte character literals and comments are recognized by the compiler. When **nombcs** or **nodbc**s is in effect, the compiler treats all literals as single-byte literals.

Syntax



Defaults

-qnombs, -qnodbcs

Usage

For rules on using multibyte characters in your source code, see "Multibyte characters" in the *XL C/C++ Language Reference*.

In addition, you can use multibyte characters in the following contexts:

- In file names passed as arguments to compiler invocations on the command line; for example:

```
ppuxlc /u/myhome/c programs/kanji files/multibyte char.c -omultibyte char
```

- In file names, as suboptions to compiler options that take file names as arguments

- In the definition of a macro name using the **-D** option; for example:

```
-DMYMACRO="kpsmultibyte_chardcs"  
-DMYMACRO='multibyte char'
```

Listing files display the date and time for the appropriate international language, and multibyte characters in the source file name also appear in the name of the corresponding list file. For example, a C source file called:

multibyte_char.c

gives a list file called

multibyte char.1st

Predefined macros

None.

Examples

To compile `myprogram.c` if it contains multibyte characters, enter:

invocation `myprogram.c -qmbcs`

Related information

- “-D” on page 80

-MF

Category

Output control

Pragma equivalent

None.

Purpose

Specifies the target for the output generated by the **-qmakedep** or **-M** options.

This option is used only together with the **-qmakedep** or **-M** options. See the description for the “-qmakedep, -M” on page 168 for more information.

Syntax

►► — **-MF***path* —————►►

Defaults

Not applicable.

Parameters

path

The target output path. *path* can be a full directory path or file name. If *path* is the name of a directory, the dependency file generated by the compiler is placed into the specified directory. If you do not specify a directory, the dependency file is stored in the current working directory.

Usage

If the file specified by **-MF** option already exists, it will be overwritten.

If you specify a single file name for the **-MF** option when compiling multiple source files, only a single dependency file will be generated containing the **make** rule for the last file specified on the command line.

Predefined macros

None.

Related information

- “-qmakedep, -M” on page 168
- “-o” on page 178
- “Directory search sequence for include files” on page 12

-qminimaltoc (PPU only)

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Controls the generation of the table of contents (TOC), which the compiler creates for an executable file in 64-bit compilation mode.

Programs compiled in 64-bit mode have a limit of 8192 TOC entries. As a result, you may encounter “relocation truncation” error messages when linking large programs in 64-bit mode; these error messages are caused by TOC overflow conditions. When **-qminimaltoc** is in effect, the compiler avoids these overflow conditions by placing TOC entries into a separate data section for each object file.

Specifying **-qminimaltoc** ensures that the compiler creates only one TOC entry for each compilation unit. Specifying this option can minimize the use of available TOC entries, but its use impacts performance. Use the **-qminimaltoc** option with discretion, particularly with files that contain frequently executed code.

Syntax

►► — -q — nominaltoc
minimaltoc —————►◄

Defaults

-qnominaltoc

Usage

This compiler option applies to 64-bit compilations only.

Compiling with **-qminimaltoc** may create slightly slower and larger code for your program. However, these effects may be minimized by specifying optimizing options when compiling your program.

Predefined macros

None.

-qmkshrobj (PPU only)

Category

Output control

Pragma equivalent

None.

Purpose

Creates a shared object from generated object files.

You should use this option, together with the related options described below, instead of calling the linker directly to create a shared object. The advantages of using this option are the automatic handling of link-time C++ template instantiation (using either the template include directory or the template registry), and compatibility with **-qipa** link-time optimizations (such as those performed at **-O5**).

Syntax

-qmkshrobj syntax

►► — **-qmkshrobj** ————— ◄◄

Defaults

By default, the output object is linked with the runtime libraries and startup routines to create an executable file.

Usage

Specifying **-qmkshrobj** implies **-qpic**.

You can also use the following related options with the **-qmkshrobj**:

-o *shared_file*

The name of the file that will hold the shared file information. The default is `a.out`.

-e *name*

Sets the entry name for the shared executable to *name*.

For detailed information on using **-qmkshrobj** to create shared libraries, see "Constructing a library" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Examples

To construct the shared library `big_lib.so` from three smaller object files, type:

invocation **-qmkshrobj -o big_lib.so lib_a.o lib_b.o lib_c.o**

Related information

- “-o” on page 178
- “-e” on page 86
- “-qpriority (C++ only)” on page 194
- “-qpic” on page 190

-qnewcheck (C++ only)

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Specifies whether the compiler inserts a check to determine whether a null pointer has been returned by a call to a version of operator new or operator new[] with a non-empty throw specification.

The check serves to guard initialization code, such as a constructor call, in case the operator new returns null. Normally, this check is not necessary, because the non-empty throw versions of operator new specify that an exception of type `std::bad_alloc` is thrown if the requested memory cannot be allocated. However, on platforms that do not support C++ exceptions, such as the Cell Broadband Engine processor SPU, operator new will only return a null pointer if memory allocation fails. When **-qnewcheck** is in effect, the compiler inserts the check, and if the null pointer is returned, the constructor for the object being allocated is not invoked. When **-qnonewcheck** is in effect, the compiler omits the check, in order to improve performance.

Syntax

►► — -q newcheck nonewcheck ◄◄

Defaults

-qnonewcheck

Usage

If you want to prevent the compiler from invoking a constructor on a null memory address (should the requested memory allocation fail) for code targeting the SPU that contains new or new[] expressions, or in code containing non-standard user-defined allocation functions, you can compile with **-qnewcheck**; however, your code will still need to handle the result of the check.

This option applies only to the `throw(std::bad_alloc)` versions of the operator new functions declared in the standard library header `<new>`, or to user-defined allocation functions with a non-empty throw specification. It does not apply to the standard or user-defined nothrow or empty throw versions of operator new, in which case the null pointer check is *always* inserted. (In the case of the inline

versions of these operators, the check is removed after optimization.) For more information, see your C++ Standard Library documentation, or the declarations in the <new> header file on your system.

Predefined macros

None.

Related information

- "The new operator" in the *XL C/C++ Language Reference*

-o

Category

Output control

Pragma equivalent

None.

Purpose

Specifies a name for the output object, assembler, or executable file.

Syntax

►► `-o` *path* ◀◀

Defaults

See "Types of output files" on page 5 for the default file names and suffixes produced by different phases of compilation.

Parameters

path

When you are using the option to compile from source files, *path* can be the name of a file or directory. The *path* can be a relative or absolute path name. When you are using the option to link from object files, *path* must be a file name.

If the *path* is the name of an existing directory, files created by the compiler are placed into that directory. If *path* is not an existing directory, the *path* is the name of the file produced by the compiler. See below for examples.

You can not specify a file name with a C or C++ source file suffix (.C, .c, .cpp, or .i), such as `myprog.c` or `myprog.i`; this results in an error and neither the compiler nor the linker is invoked.

Usage

If you use the `-c` option with `-o` together and the *path* is not an existing directory, you can only compile one source file at a time. In this case, if more than one source file name is listed in the compiler invocation, the compiler issues a warning message and ignores `-o`.

The **-E**, **-P**, and **-qsyntaxonly** options override the **-o** option.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the resulting executable is called `myaccount`, assuming that no directory with name `myaccount` exists, enter:

```
invocation myprogram.c -o myaccount
```

To compile `test.c` to an object file only and name the object file `new.o`, enter:

```
invocation test.c -c -o new.o
```

Related information

- “-c” on page 65
- “-E” on page 87
- “-P” on page 184
- “-qsyntaxonly (C only)” on page 235

-O, -qoptimize

Category

Optimization and tuning

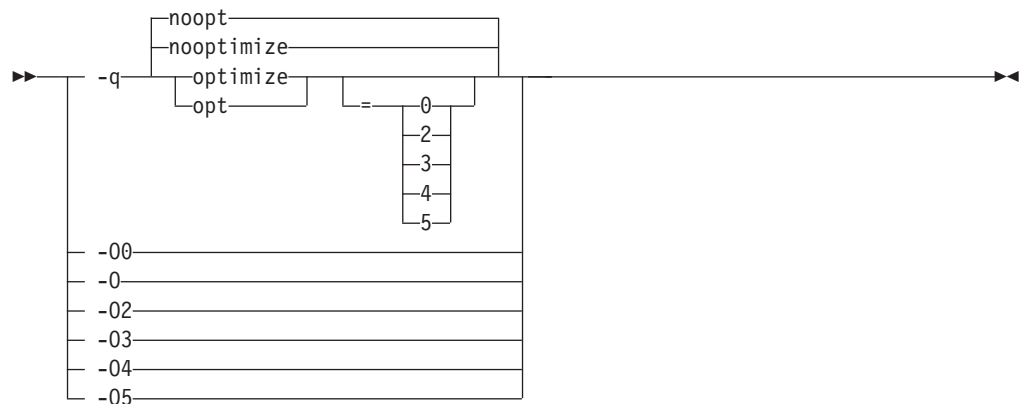
Pragma equivalent

#pragma options [no]optimize

Purpose

Specifies whether to optimize code during compilation and, if so, at which level.

Syntax



Defaults

-qnooptimize or **-O0** or **-qoptimize=0**

Parameters

-O0 | nooptimize | noopt | optimize | opt=0

Performs only quick local optimizations such as constant folding and elimination of local common subexpressions.

This setting implies **-qstrict_induction** unless **-qnostrict_induction** is explicitly specified.

-O | -O2 | optimize | opt | optimize | opt=2

Performs optimizations that the compiler developers considered the best combination for compilation speed and runtime performance. The optimizations may change from product release to release. If you need a specific level of optimization, specify the appropriate numeric value.

This setting implies **-qstrict** and **-qnostrict_induction**, unless explicitly negated by **-qstrict_induction** or **-qnostrict**.

-O3 | optimize | opt=3

Performs additional optimizations that are memory intensive, compile-time intensive, or both. They are recommended when the desire for runtime improvement outweighs the concern for minimizing compilation resources.

-O3 applies the **-O2** level of optimization, but with unbounded time and memory limits. **-O3** also performs higher and more aggressive optimizations that have the potential to slightly alter the semantics of your program. The compiler guards against these optimizations at **-O2**. The aggressive optimizations performed when you specify **-O3** are:

1. Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed.

Loads and floating-point computations fall into this category. This optimization is aggressive because it may place such instructions onto execution paths where they *will* be executed when they *may* not have been according to the actual semantics of the program.

For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved at **-O2** because the computation may cause an exception. At **-O3**, the compiler will move it because it is not certain to cause an exception. The same is true for motion of loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable at **-O3**. Loads in general are not considered to be absolutely safe at **-O2** because a program can contain a declaration of a static array `a` of 10 elements and load `a[600000000003]`, which could cause a segmentation violation.

The same concepts apply to scheduling.

Example:

In the following example, at **-O2**, the computation of `b+c` is not moved out of the loop for two reasons:

- It is considered dangerous because it is a floating-point operation
- `t` does not occur on every path through the loop

At **-O3**, the code is moved.

```
...
int i ;
float a[100], b, c ;
for (i = 0 ; i < 100 ; i++)
{
```

```

    if (a[i] < a[i+1])
        a[i] = b + c ;
    }
    ...

```

2. Conformance to IEEE rules are relaxed.

With **-O2** certain optimizations are not performed because they may produce an incorrect sign in cases with a zero result, and because they remove an arithmetic operation that may cause some type of floating-point exception.

For example, $X + 0.0$ is not folded to X because, under IEEE rules, $-0.0 + 0.0 = 0.0$, which is $-X$. In some other cases, some optimizations may perform optimizations that yield a zero result with the wrong sign. For example, $X - Y * Z$ may result in a -0.0 where the original computation would produce 0.0 .

In most cases the difference in the results is not important to an application and **-O3** allows these optimizations.

3. Floating-point expressions may be rewritten.

Computations such as $a*b*c$ may be rewritten as $a*c*b$ if, for example, an opportunity exists to get a common subexpression by such rearrangement. Replacing a divide with a multiply by the reciprocal is another example of reassociating floating-point computations.

4. When compiling a program that targets the PPU using one of the ppu invocation commands, such as **ppuxlc**, specifying **-O3** implies **-qhot=level=0**, unless you explicitly specify **-qhot** or **-qhot=level=1**. When compiling a program that targets the SPU using one of the spu invocation commands, such as **spuxlc**, specifying **-O3** implies **-qhot**. When compiling a program that targets both PPU and SPU using one of the cbe invocation commands, such as **cbexlc**, **-qsmp** is on by default and implies **-qhot**.

-qfloat=rsqrt is set by default with **-O3**.

-qmaxmem=1 is set by default with **-O3**, allowing the compiler to use as much memory as necessary when performing optimizations.

Built-in functions do not change `errno` at **-O3**.

Integer divide instructions are considered too dangerous to optimize even at **-O3**.

Refer to “**-qflttrap** (PPU only)” on page 103 to see the behavior of the compiler when you specify **optimize** options with the **-qflttrap** option.

You can use the **-qstrict** and **-qstrict_induction** compiler options to turn off effects of **-O3** that might change the semantics of a program. Specifying **-qstrict** together with **-O3** invokes all the optimizations performed at **-O2** as well as further loop optimizations. Reference to the **-qstrict** compiler option can appear before or after the **-O3** option.

The **-O3** compiler option followed by the **-O** option leaves **-qignerrno** on.

When **-O3** and **-qhot=level=1** are in effect, the compiler replaces any calls in the source code to standard math library functions with calls to the equivalent MASS library functions, and if possible, the vector versions.

-O4 | optimize | opt=4

This option is the same as **-O3**, except that it also:

- Sets the **-qhot** option
- Sets the **-qipa** option

Note: Later settings of **-O**, **-qcache**, **-qhot**, **-qipa**, **-qarch**, and **-qtune** options will override the settings implied by the **-O4** option.

-O5 | optimize | opt=5

This option is the same as **-O4**, except that it:

- Sets the **-qipa=level=2** option to perform full interprocedural data flow and alias analysis.

Note:

- Later settings of **-O**, **-qcache**, **-qipa**, **-qarch**, and **-qtune** options will override the settings implied by the **-O5** option.
- The use of **-O5** compiler options is recommended on the SPU to get the maximum performance from your application.

Usage

Increasing the level of optimization may or may not result in additional performance improvements, depending on whether additional analysis detects further opportunities for optimization.

Compilations with optimizations may require more time and machine resources than other compilations.

Optimization can cause statements to be moved or deleted, and generally should not be specified along with the **-g** flag for debugging programs. The debugging information produced may not be accurate.

Predefined macros

- `__OPTIMIZE__` is predefined to 2 when **-O | O2** is in effect; it is predefined to 3 when **-O3 | O4 | O5** is in effect. Otherwise, it is undefined.
- `__OPTIMIZE_SIZE__` is predefined to 1 when **-O | -O2 | -O3 | -O4 | -O5** and **-qcompact** are in effect. Otherwise, it is undefined.

Examples

To compile and optimize `myprogram.c`, enter:

```
invocation myprogram.c -O3
```

Related information

- “**-qhot**” on page 115
- “**-qipa**” on page 132
- “**-qpdf1**, **-qpdf2** (PPU only)” on page 187
- “**-qstrict**” on page 227
- “Optimizing your applications” in the *XL C/C++ Optimization and Programming Guide*.

-qoptdebug

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

When used with high levels of optimization, produces files containing optimized pseudocode that can be read by a debugger.

An output file with a .optdbg extension is created for each source file compiled with **-qoptdebug**. You can use the information contained in this file to help you understand how your code actually behaves under optimization.

Syntax

►► `-q` nooptdebug
optdebug _____ ►►

Defaults

`-qnooptdebug`

Usage

-qoptdebug only has an effect when used with an option that enables the high-level optimizer, namely **-O3** or higher optimization level, or **-qhot**, **-qipa**, or **-qpdf**. You can use the option on both compilation and link steps. If you specify it on the compile step, one output file is generated for each source file. If you specify it on the **-qipa** link step, a single output file is generated.

You must still use the **-g** or **-qlinedebug** option to include debugging information that can be used by a debugger.

For more information and examples of using this option, see "Using `-qoptdebug` to help debug optimized programs" in the *XL C/C++ Optimization and Programming Guide*.

Related information

- "`-O`, `-optimize`" on page 179
- "`-qhot`" on page 115
- "`-qipa`" on page 132
- "`-qpdf1`, `-qpdf2` (PPU only)" on page 187
- "`-g`" on page 108
- "`-qlinedebug`" on page 163

-p, -pg, -qprofile (PPU only)

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Prepares the object files produced by the compiler for profiling.

When you compile with a profiling option, the compiler produces monitoring code that counts the number of times each routine is called. The compiler replaces the startup routine of each subprogram with one that calls the monitor subroutine at the start. When you execute the compiled program and it ends normally, it writes the recorded information to a gmon.out file. You can then use the **gprof** command to generate a runtime profile.

Syntax



Defaults

Not applicable.

Usage

When you are compiling and linking in separate steps, you must specify the profiling option in both steps.

If the **-qtbtable** option is not set, the profiling options will generate full traceback tables.

Predefined macros

None.

Examples

To compile myprogram.c to include profiling data, enter:

```
invocation myprogram.c -p
```

Remember to compile *and* link with one of the profiling options. For example:

```
invocation myprogram.c -p -c
invocation myprogram.o -p -o program
```

Related information

- “-qtbtable (PPU only)” on page 238
- See your operating system documentation for more information on the **gprof** command.

-P

Category

Output control

Pragma equivalent

None.

Purpose

Preprocesses the source files named in the compiler invocation, without compiling, and creates an output preprocessed file for each input file.

The preprocessed output file has the same name as the input file, with an `.i` suffix.

Syntax

►— `-P` —►

Defaults

By default, source files are preprocessed, compiled, and linked to produce an executable file.

Usage

The `-P` option accepts any file name, except those with an `.i` suffix. Otherwise, source files with unrecognized file name suffixes are treated and preprocessed as C files, and no error message is generated.

Unless `-qppline` is specified, `#line` directives are not generated.

Line continuation sequences are removed and the source lines are concatenated.

The `-P` option retains all white space including line-feed characters, with the following exceptions:

- All comments are reduced to a single space (unless `-C` is specified).
- Line feeds at the end of preprocessing directives are not retained.
- White space surrounding arguments to function-style macros is not retained.

The `-P` option is overridden by the `-E` option. The `-P` option overrides the `-c`, `-o`, and `-qsyntaxonly` option.

Predefined macros

None.

Related information

- “`-C`, `-C!`” on page 66
- “`-E`” on page 87
- “`-qppline`” on page 191
- “`-qsyntaxonly` (C only)” on page 235

-qpath

Category

Compiler customization

Pragma equivalent

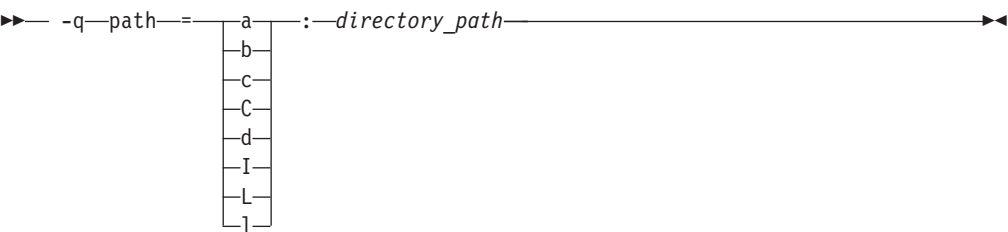
None.

Purpose

Determines substitute path names for XL C/C++ executables such as the compiler, assembler, and linker.

You can use this option if you want to keep multiple levels of some or all of the XL C/C++ executables and have the option of specifying which one you want to use. This option is preferred over the **-B** and **-t** options.

Syntax



Defaults


By default, the compiler uses the paths for compiler components defined in the configuration file.

Parameters

directory_path

The path to the directory where the alternate programs are located.

The following table shows the correspondence between **-qpath** parameters and the component executable names:

Parameter	Description	Executable name
a	Assembler	ppu-as or spu-as
b	Low-level optimizer	xlCcode
c	Compiler front end	xlcentry, xlCentry
 C	C++ compiler front end	xlCentry
d	Disassembler	dis
I	High-level optimizer, compile step	ipa
L	High-level optimizer, link step	ipa
l	Linker	ppu-ld or spu-ld

Usage

The **-qpath** option overrides the **-F**, **-t**, and **-B** options.

Predefined macros

None.

Examples

To compile `myprogram.c` using a substitute `xl` compiler in `/lib/tmp/mine/` enter:

```
invocation myprogram.c -qpath=c:/lib/tmp/mine/
```

To compile `myprogram.c` using a substitute linker in `/lib/tmp/mine/`, enter:

```
invocation myprogram.c -qpath=l:/lib/tmp/mine/
```

Related information

- “-B” on page 63
- “-F” on page 94
- “-t” on page 236

-qpdf1, -qpdf2 (PPU only)

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

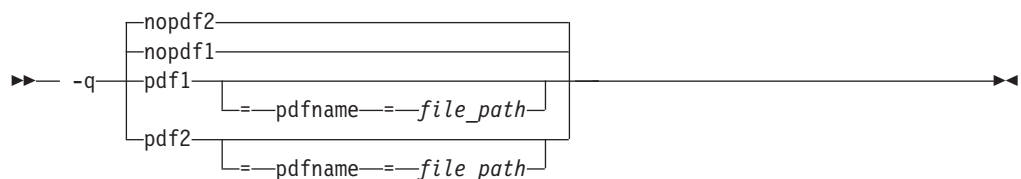
Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.

Note: You can only use `qpdf` with one of the `ppu` invocation commands, such as `ppuxlc`. You can not use this option for a program that targets both PPU and SPU and that is compiled using one of the `cbe` invocation commands, such as `cbexlc`.

PDF is a two-step process. You first compile the application with `-qpdf1` and a minimum optimization level of `-O2`, with linking. You then run the resulting application with a typical data set. During the test run, profile data is written to a profile file (by default, this file is named `._pdf` and is saved in the current working directory, or in the directory named by the `PDFDIR` environment variable, if it is set). You then recompile, and/or link or relink the application with `-qpdf2` and a minimum optimization level of `-O2`, which fine-tunes the optimizations applied according to the profile data collected during the program execution.

PDF is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production.

Syntax



Defaults

-qnopdf1, -qnopdf2

Parameters

pdfname= *file_path*

Specifies the path to the file that will hold the profile data. By default, the file name is `._pdf`, and it is placed in the current working directory or in the directory named by the PDFDIR environment variable. You can use the **pdfname** suboption to allow you to do simultaneous runs of multiple executables using the same PDF directory. This is especially useful when tuning with PDF on dynamic libraries.

Usage

You must compile the main program with PDF for profiling information to be collected at run time.

You must use the same compilation options for the PDF2 compilation as for the PDF1 compilation.

If you do not want the optimized object files to be relinked during the second step, specify **-qpdf2 -qnoipa**. Note, however, that if you change a source file that was compiled previously with **-qpdf1**, you will need to go through the entire first pass process again.

To specify the directory and file name for the profile file when using both **-qpdf1** and **-qpdf2**, you must use the **pdfname** suboption. For examples, see "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*.

For recommended procedures for using PDF, see "Using profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide*.

The following utility programs, found in `/opt/ibmcmp/xlc/cbe/10.1/bin/`, are available for managing the directory to which profile data is written:

cleanpdf

►► cleanpdf *directory_path* ►►

Removes all profiling information from the directory specified by *directory_path*; or if *pathname* is not specified, from the directory set by the PDFDIR environment variable; or if PDFDIR is not set, from the current directory. Removing profiling information reduces runtime overhead if you change the program and then go through the PDF process again.

Run **cleanpdf** only when you are finished with the PDF process for a particular application. Otherwise, if you want to resume using PDF with that application, you will need to recompile all of the files again with **-qpdf1**.

resetpdf

►►—resetpdf—
└──────────┐
 directory_path

Same as **cleanpdf**, described above.

Predefined macros

None.

Related information

- “-qipa” on page 132
- “Optimizing your applications” in the *XL C/C++ Optimization and Programming Guide*

-qphsinfo

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Reports the time taken in each compilation phase to standard output.

Syntax

►►—-q—
 └───┐
 nophsinfo
 phsinfo

Defaults

-qnophsinfo

Usage

The output takes the form *number1/number2* for each phase where *number1* represents the CPU time used by the compiler and *number2* represents the total of the compiler time and the time that the CPU spends handling system calls.

Predefined macros

None.

Examples

 To compile `myprogram.c` and report the time taken for each phase of the compilation, enter:

invocation `myprogram.c -qphsinfo`

The output will look similar to:

C Init	- Phase Ends;	0.010/	0.040
IL Gen	- Phase Ends;	0.040/	0.070
W-TRANS	- Phase Ends;	0.000/	0.010
OPTIMIZ	- Phase Ends;	0.000/	0.000
REGALLO	- Phase Ends;	0.000/	0.000
AS	- Phase Ends;	0.000/	0.000

Compiling the same program with **-O4** gives:

C Init	- Phase Ends;	0.010/	0.040
IL Gen	- Phase Ends;	0.060/	0.070
IPA	- Phase Ends;	0.060/	0.070
IPA	- Phase Ends;	0.070/	0.110
W-TRANS	- Phase Ends;	0.060/	0.180
OPTIMIZ	- Phase Ends;	0.010/	0.010
REGALLO	- Phase Ends;	0.010/	0.020
AS	- Phase Ends;	0.000/	0.000



To compile myprogram.C and report the time taken for each phase of the compilation, enter:

```
ppuxlc++ myprogram.C -qphsinfo
```

The output will look similar to:

Front End	- Phase Ends;	0.004/	0.005
W-TRANS	- Phase Ends;	0.010/	0.010
OPTIMIZ	- Phase Ends;	0.000/	0.000
REGALLO	- Phase Ends;	0.000/	0.000
AS	- Phase Ends;	0.000/	0.000

Compiling the same program with **-O4** gives:

Front End	- Phase Ends;	0.004/	0.006
IPA	- Phase Ends;	0.040/	0.040
IPA	- Phase Ends;	0.220/	0.280
W-TRANS	- Phase Ends;	0.030/	0.110
OPTIMIZ	- Phase Ends;	0.030/	0.030
REGALLO	- Phase Ends;	0.010/	0.050
AS	- Phase Ends;	0.000/	0.000

-qpic

Category

Object code control

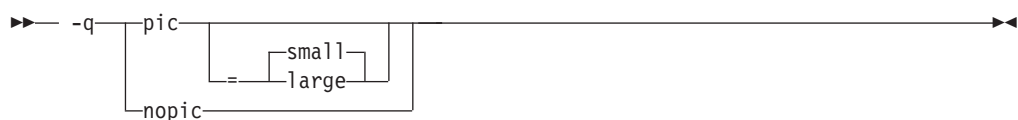
Pragma equivalent

None.

Purpose

Generates Position-Independent Code suitable for use in shared libraries.

Syntax



Defaults

- **-qnopic** for 32-bit compilation targeting the PPU or SPU.
- **-qplic=small** for 64-bit compilation targeting the PPU.
- **-qplic=small** when the **-qmkshrobj** compiler option is specified.

Parameters

small

Instructs the compiler to assume that the size of the Global Offset Table in 32-bit mode or Table of Contents in 64-bit mode is no larger than 64 Kb.

large

Allows the Global Offset Table to be larger than 64 Kb in size, allowing more addresses to be stored in the table. Code generated with this option is usually larger than that generated with **-qplic=small**. **-qplic=large** only works in 32-bit mode.

Specifying **-qplic** without any suboptions is equivalent to **-qplic=small**.

Usage

For 64-bit compilation targeting the PPU, **-qplic** is enabled and cannot be disabled.

Predefined macros

None.

Examples

To compile a shared library `libmylib.so`, use the following commands:

```
invocation mylib.c -qplic=small -c -o mylib.o
invocation -qmkshrobj mylib -o libmylib.so.1
```

Related information

- “-q32, -q64 (PPU only)” on page 49
- “-qmkshrobj (PPU only)” on page 176

-qpplline

Category

Object code control

Pragma equivalent

None.

Purpose

When used in conjunction with the **-E** or **-P** options, enables or disables the generation of `#line` directives.

Syntax

►► — `-qpplline` —►►
 └─ `-nopplline` ─┘

Defaults

- **-qnoppline** when **-P** is in effect
- **-qppline** when **-E** is in effect

Usage

The **-C** option has no effect without either the **-E** or the **-P** option. With the **-E** option, line directives are written to standard output. With the **-P** option, line directives are written to an output file.

Predefined macros

None.

Examples

To preprocess `myprogram.c` to write the output to `myprogram.i`, and generate `#line` directives:

invocation `myprogram.c -P -qppline`

Related information

- “**-E**” on page 87
- “**-P**” on page 184

-qprefetch (PPU only)

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Inserts prefetch instructions automatically where there are opportunities to improve code performance.

When **-qprefetch** is in effect, the compiler may insert prefetch instructions in compiled code. When **-qnoprefetch** is in effect, prefetch instructions are not inserted in compiled code.

Syntax



Defaults

-qprefetch

Usage

The **-qnoprefetch** option will not prevent built-in functions such as `__prefetch_by_stream` from generating prefetch instructions.

Predefined macros

None.

-qprint

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Enables or suppresses listings.

When **-qprint** is in effect, listings are enabled if they are requested by other compiler options that produce listings. When **-qnoprint** is in effect, all listings are suppressed, regardless of whether listing-producing options are specified.

Syntax



Defaults

-qprint

Usage

You can use **-qnoprint** to override all listing-producing options and equivalent pragmas, regardless of where they are specified. These options are:

- **-qattr**
- **-qlist**
- **-qlistopt**
- **-qsource**
- **-qxref**

Predefined macros

None.

Examples

To compile `myprogram.c` and suppress all listings, even if some files have **#pragma options source** and similar directives, enter:

invocation myprogram.c -qnoprint

-qpriority (C++ only)

Category

Object code control

Pragma equivalent

#pragma options priority, #pragma priority

Purpose

Specifies the priority level for the initialization of static objects.

The C++ standard requires that all global objects within the same translation unit be constructed from top to bottom, but it does not impose an ordering for objects declared in different translation units. The **-qpriority** option and **#pragma priority** directive allow you to impose a construction order for all static objects declared within the same load module. Destructors for these objects are run in reverse order during termination.

Syntax

Option syntax

►► — -q—priority—==—*number*—————►►

Pragma syntax

►► —#—pragma—priority—(—*number*—)—————►►

Defaults

The default priority level is 65 535.

Parameters

number

An integer literal in the range of 101 to 65 535. A lower value indicates a higher priority; a higher value indicates a lower priority. If you do not specify a *number*, the compiler assumes 65 535.

Usage

More than one **#pragma priority** can be specified within a translation unit. The priority value specified in one pragma applies to the constructions of all global objects declared after this pragma and before the next one. However, in order to be consistent with the Standard, priority values specified within the same translation unit must be strictly increasing. Objects with the same priority value are constructed in declaration order.

The effect of a **#pragma priority** exists only within one load module. Therefore, **#pragma priority** cannot be used to control the construction order of objects in different load modules.

Note: The C++ variable attribute `init_priority` can also be used to assign a priority level to a shared variable of class type. See "The `init_priority` variable attribute" in the *XL C/C++ Language Reference* for more information.

Examples

To compile the file `myprogram.C` to produce an object file `myprogram.o` so that objects within that file have an initialization priority of 2 000, enter:

```
ppuxlc++ myprogram.C -c -qpriority=2000
```

Related information

- "Initializing static objects in libraries" in the *XL C/C++ Optimization and Programming Guide*

-qprocimported, -qproclocal, -qprocunknown (PPU only)

Category

Optimization and tuning

Pragma equivalent

#pragma options `proclocal`, #pragma options `procimported`, #pragma options `procunknown`

Purpose

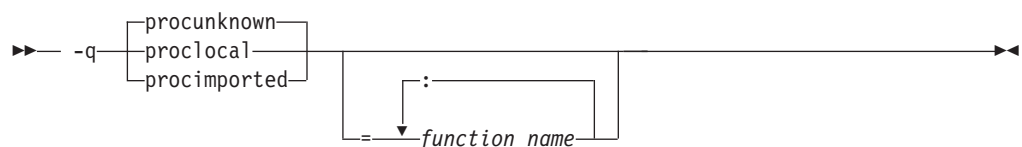
Marks functions as local, imported, or unknown in 64-bit compilations.

Local functions are statically bound with the functions that call them; smaller, faster code is generated for calls to such functions. You can use the **proclocal** option or pragma to name functions that the compiler can assume are local.

Imported functions are dynamically bound with a shared portion of a library. Code generated for calls to functions marked as imported may be larger, but is faster than the default code sequence generated for functions marked as unknown. You can use the **procimported** option or pragma to name functions that the compiler can assume are imported.

Unknown functions are resolved to either statically or dynamically bound objects during linking. You can use the **procunknown** option or pragma to name functions that the compiler can assume are unknown.

Syntax



Defaults

-qprocunknown: The compiler assumes that all functions' definitions are unknown.

Parameters

function_name

The name of a function that the compiler should assume is local, imported, or unknown (depending on the option specified). If you do not specify any *function_name*, the compiler assumes that *all* functions are local, imported, or unknown.

▶ **C++** Names must be specified using their mangled names. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file. (See also "Name mangling" in the *XL C/C++ Language Reference* for details on using the `extern "C"` linkage specifier on declarations to prevent name mangling.)

Usage

This option applies to 64-bit compilations only.

If any functions that are marked as local resolve to shared library functions, the linker will detect the error and issue warnings. If any of the functions that are marked as imported resolve to statically bound objects, the generated code may be larger and run more slowly than the default code sequence generated for unknown functions.

If you specify more than one of these options with no function names, the last option specified is used. If you specify the same function name on more than one option specification, the last one is used.

Predefined macros

None.

Examples

To compile `myprogram.c` along with the archive library `oldprogs.a` so that:

- Functions `fun` and `sun` are specified as local
- Functions `moon` and `stars` are specified as imported
- Function `venus` is specified as unknown

use the following command:

```
invocation myprogram.c oldprogs.a -qprolocal=fun(int):sun()  
          -qprocimported=moon():stars(float) -qprocunknown=venus()
```

If the following example, in which a function marked as local instead resolves to a shared library function, is compiled with **-qprocllocal**:

```
int main(void)  
{  
    printf("Just in function fool()\n");  
    printf("Just in function fool()\n");  
}
```

a linker error will result. To correct this problem, you should explicitly mark the called routine as being imported from a shared object. In this case, you would recompile the source file and explicitly mark `printf` as imported by compiling with `-qproclcal -qprocimported=printf`.

Related information

- “-qdataimported, -qdatalocal, -qtocdata (PPU only)” on page 81

-qproto (C only)

Category

Object code control

Pragma equivalent

#pragma options [no]proto

Purpose

Specifies the linkage conventions for passing floating-point arguments to functions that have not been prototyped.

When **proto** is in effect, the compiler assumes that the arguments in function calls are the same types as the corresponding parameters of the function definition, even if the function has not been prototyped. By asserting that an unprototyped function actually expects a floating-point argument if it is called with one, you allow the compiler to pass floating-point arguments in floating-point registers exclusively. When **noproto** is in effect, the compiler does not make this assumption, and must pass floating-point parameters in floating-point and general purpose registers.

Syntax

```
➡ — -q — [noproto] — [proto] — ➡
```

Defaults

-qnoproto

Usage

This option only is only valid when the compiler allows unprototyped functions; that is, with the **ppucc**, **spucc**, **ppuxlc**, or **spuxlc** invocation command, or with the **-qlanglvl** option set to **classic** | **extended** | **extc89** | **extc99**.

Predefined macros

None.

Examples

To compile `my_c_program.c` to allow the compiler to use the standard linkage conventions for floating-point parameters, even when functions are not prototyped, enter:

invocation my_c_program.c -qproto

-Q, -qinline

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Attempts to inline functions instead of generating calls to those functions, for improved performance.

C++ Specifying **-Q** (or **-qinline**) enables automatic inlining by the compiler front end. Specifying **-Q** with **-O** provides additional inlining by enabling inlining by the low-level optimizer. In both cases, the compiler attempts to inline all functions, in addition to those defined inside a class declaration or explicitly marked with the `inline` specifier.

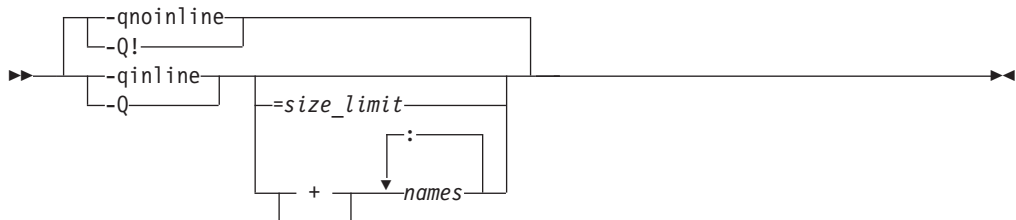
C You must specify a minimum optimization level of **-O** along with **-Q** (or **-qinline**) to enable inlining of functions, including those declared with the `inline` specifier. You can also use the **-Q** option to specify restrictions on the functions that should or should not be inlined.

In all cases where **-Q** (or **-qinline**) is in effect, the compiler uses heuristics to determine whether inlining a specific function will result in a performance benefit. That is, whether a function is appropriate for inlining is subject to limits on the number of inlined calls and the amount of code size increase as a result. Therefore, simply enabling inlining does not guarantee that a given function will be inlined.

Specifying **-Q!** (or **-qnoinline**) disables all inlining, including that performed by the high-level optimizer with the **-qipa** option, and functions declared explicitly as `inline`.

Syntax

-qinline and -Q syntax — C



-qinline and -Q syntax — C++



Defaults

-qnoinline or -Q!

Parameters

► **C** *size_limit*

A positive integer representing the number of executable statements in a function. Declarations are not counted, as you can see in the example below:

```
increment()
{
    int a, b, i;
    for (i=0; i<10; i++) /* statement 1 */
    {
        a=i;             /* statement 2 */
        b=i;             /* statement 3 */
    }
}
```

The number of executable statements in a function must be fewer than or equal to *size_limit* for it to be considered for inlining. Specifying a value of 0 causes no functions to be inlined except those functions listed in the *name* suboption, or those marked with supported forms of the `inline` function specifier. If you do not specify *size*, the default value is 20.

► **C** +

The compiler attempts to inline all functions that meet the criterion specified by *size*, as well as those listed by *name*.

► **C** -

The compiler attempts to inline all functions that meet the criterion specified by *size*, except those listed by *name*.

► **C** *name*

The name of a function to be inlined. Separate each function name with a colon (:). This suboption overrides any setting of the *size* value. Note that this suboption does not affect functions explicitly declared with the `inline` specifier; when **-O** and **-Q** | **-qinline** are in effect, those functions are *always* considered for inlining. You can specify this suboption as an argument to both the positive and negative forms of the options, to precisely control which functions are most likely to be inlined.

A warning message is issued for functions that are not defined in the source file being compiled.

Usage

To maximize inlining, specify optimization (**-O**) and also specify the appropriate **-qinline** or **-Q** options.

Because inlining does not always improve runtime performance, you should test the effects of this option on your code. Do not attempt to inline recursive or mutually recursive functions.

If you specify the **-g** option to generate debug information, inlining may be suppressed.


Predefined macros

None.


Examples

To compile `myprogram.c` so that no functions are inlined, enter:


```
invocation myprogram.c -O -qnoinline
```

 To compile `myprogram.c` so that the compiler attempts to inline all functions of fewer than 12 statements, enter:


```
invocation myprogram.c -O -qinline=12
```

 Assuming that the functions `salary`, `taxes`, `expenses`, and `benefits` have more than 20 executable statements each, to compile `myprogram.c` so that the compiler attempts to inline all appropriate functions (that is, those that have fewer than the default of 20 statements) *plus* these functions, enter:

```
invocation myprogram.c -O -qinline+salary:taxes:expenses:benefits
```

 Assuming that the functions `salary`, `taxes`, `expenses`, and `benefits` have fewer than 20 executable statements each, to compile `myprogram.c` so that the compiler attempts to inline all appropriate functions (that is, those that have fewer than the default of 20 statements) *except* these functions, enter:

```
invocation myprogram.c -O -qinline-salary:taxes:expenses:benefits
```

 You can use a size value of zero along with function names to inline specific functions. For example:

```
-O -qinline=0
```

followed by:

```
-qinline+salary:taxes:benefits
```

causes *only* the functions named `salary`, `taxes`, or `benefits` to be inlined, if possible, and no others.

Related information

- “-g” on page 108
- “-qipa” on page 132
- “-O, -qoptimize” on page 179
- “The inline function specifier” in the *XL C/C++ Language Reference*

-r

Category

Object code control

Pragma equivalent

None.

Purpose

Produces a relocatable object, even though the file contains unresolved symbols.

Syntax

►► — -r ————— ►►

Defaults

Not applicable.

Usage

A file produced with this flag is expected to be used as a file parameter in another compiler invocation.

Predefined macros

None.

Examples

To compile `myprogram.c` and `myprog2.c` into a single object file `mytest.o`, enter:
invocation `myprogram.c myprog2.c -r -o mytest.o`

-R

Category

Linking

Pragma equivalent

None.

Purpose

At link time, writes search paths for shared libraries into the executable, so that these directories are searched at program run time for any required shared libraries.

Syntax

►► — -R—*directory_path*————— ►►

Defaults

The default is to include only the standard directories. See the compiler configuration file for the directories that are set by default.

Usage

If the **-R***directory_path* option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first at run time.

The **-R** compiler option is cumulative. Subsequent occurrences of **-R** on the command line do not replace, but add to, any directory paths specified by earlier occurrences of **-R**.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the directory `/usr/tmp/old` is searched at run time along with standard directories for the dynamic library `libspfiles.so`, enter:

```
invocation myprogram.c -lspfiles -R/usr/tmp/old
```

Related information

- “-L” on page 147

-qreport

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Produces listing files that show how sections of code have been optimized.

A listing file is generated with a `.lst` suffix for each source file named on the command line. When used with an option that enables vectorization, the listing file shows a pseudo-C code listing and a summary of how program loops are optimized. The report also includes diagnostic information to show why specific loops could not be vectorized. For instance, when used with **-qhot=simd**, messages are provided to identify non-stride-one references that may prevent loop vectorization.

The compiler also reports the number of streams created for a given loop, which include both load and store streams. This information is included in the Loop Transformation section of the listing file. You can use this information to understand your application code and to tune your code for better performance. For example, you can distribute a loop which has more streams than the number supported by the underlying architecture. POWER4 and POWER5 support load stream prefetch and POWER6 supports both load and store stream prefetch.

Syntax

►► — -q — noreport
report —————►►

Defaults

-qnoreport

Usage

For **-qreport** to generate a loop transformation listing, you must also specify one of the following options on the command line:

- **-qhot[=simd]**
- **-O5**
- **-qipa=level=2**

If you use **-qreport** with **-O5** or **-qipa=level=2**, the report will be generated after the link step.

The pseudo-C code listing is not intended to be compilable. Do not include any of the pseudo-C code in your program, and do not explicitly call any of the internal routines whose names may appear in the pseudo-C code listing.

Predefined macros

None.

Examples

To compile `myprogram.c` so the compiler listing includes a report showing how loops are optimized, enter:

invocation `-qhot -O3 -qreport myprogram.c`

Related information

- “-qhot” on page 115
- “-qipa” on page 132
- “-qoptdebug” on page 182
- “Using -qoptdebug to help debug optimized programs” in the *XL C/C++ Optimization and Programming Guide*

-qreserved_reg

Category

Object code control

Pragma equivalent

None.

Purpose

Indicates that the given list of registers cannot be used during the compilation except as a stack pointer, frame pointer or in some other fixed role.

You should use this option in modules that are required to work with other modules that use global register variables or hand-written assembler code.

Syntax

►► — -q-reserved_reg=*register_name* —►►



Defaults

Not applicable.

Parameters

register_name

A valid register name on the target platform. Valid registers for the PPU are:

r0 to r31

General purpose registers

f0 to f31

Floating-point registers

v0 to v31

Vector registers

Valid registers for the SPU are 0 to 127, or v0 to v127.

Usage

-qreserved_reg is cumulative, for example, specifying **-qreserved_reg=r14** and **-qreserved_reg=r15** is equivalent to specifying **-qreserved_reg=r14:r15**.

Duplicate register names are ignored.

Predefined macros

None.

Examples

To specify that `myprogram.c` reserves the general purpose registers `r3` and `r4`, enter:

invocation `myprogram.c -qreserved_reg=r3:r4`

Related information

- "Variables in specified registers" in the *XL C/C++ Language Reference*

-qro (PPU only)

Category

Object code control

Pragma equivalent

`#pragma options ro, #pragma strings`

Purpose

Specifies the storage type for string literals.

When **ro** or **strings=readonly** is in effect, strings are placed in read-only storage. When **norow** or **strings=writeable** is in effect, strings are placed in read/write storage.

Syntax

Option syntax

►► `-q` ro
norow _____►►

Pragma syntax

►► `#pragma strings` readonly
writeable `()` _____►►

Defaults

C Strings are read-only for all invocation commands except **ppucc** or **spucc**. If the **ppucc** or **spucc** invocation command is used, strings are writeable.

C++ Strings are read-only.

Parameters

readonly (pragma only)

String literals are to be placed in read-only memory.

writeable (pragma only)

String literals are to be placed in read-write memory.

Usage

Placing string literals in read-only memory can improve runtime performance and save storage. However, code that attempts to modify a read-only string literal may generate a memory error.

The pragmas must appear before any source statements in a file.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the storage type is writable, enter:

invocation `myprogram.c -qnoro`

Related information

- “-qro (PPU only)” on page 204
- “-qroconst (PPU only)”

-qroconst (PPU only)

Category

Object code control

Pragma equivalent

#pragma options [no]roconst

Purpose

Specifies the storage location for constant values.

When **roconst** is in effect, constants are placed in read-only storage. When **norconst** is in effect, constants are placed in read/write storage.

Syntax

►► -q roconst
norconst _____ ►►

Defaults

- C **-qroconst** for all compiler invocations except **ppucc** or **spucc** and its derivatives. **-qnoroconst** for the **ppucc** or **spucc** invocation and its derivatives.
- C++ **-qroconst**

Usage

Placing constant values in read-only memory can improve runtime performance, save storage, and provide shared access. However, code that attempts to modify a read-only constant value generates a memory error.

“Constant” in the context of the **-qroconst** option refers to variables that are qualified by **const**, including **const**-qualified characters, integers, floats, enumerations, structures, unions, and arrays. The following constructs are not affected by this option:

- Variables qualified with **volatile** and aggregates (such as a structure or a union) that contain **volatile** variables
- Pointers and complex aggregates containing pointer members
- Automatic and static types with block scope
- Uninitialized types
- Regular structures with all members qualified by **const**
- Initializers that are addresses, or initializers that are cast to non-address values

The **-qroconst** option does not imply the **-qro** option. Both options must be specified if you wish to specify storage characteristics of both string literals (**-qro**) and constant values (**-qroconst**).

Predefined macros

None.

Related information

- “-gro (PPU only)” on page 204

-qrtti (C++ only) (PPU only)

Category

Object code control

Pragma equivalent

#pragma options rtti

Purpose

Generates runtime type identification (RTTI) information for exception handling and for use by the typeid and dynamic_cast operators.

Syntax

►► -q rtti
nortti ◀◀

Defaults

-qrtti

Usage

For improved runtime performance, suppress RTTI information generation with the **-qnortti** setting.

You should be aware of the following effects when specifying the **-qrtti** compiler option:

- Contents of the virtual function table will be different when **-qrtti** is specified.
- When linking objects together, all corresponding source files must be compiled with the correct **-qrtti** option specified.
- If you compile a library with mixed objects (**-qrtti** specified for some objects, **-qnortti** specified for others), you may get an undefined symbol error.

Predefined macros

- `__RTTI_ALL__` is defined to 1 when **-qrtti** is in effect; otherwise, it is undefined.
- `__NO_RTTI__` is defined to 1 when **-qnortti** is in effect; otherwise, it is undefined.

Related information

- “-qeh (C++ only) (PPU only)” on page 89

-S

Category

Object code control

Pragma equivalent

None.

Purpose

Strips the symbol table, line number information, and relocation information from the output file.

This command is equivalent to the operating system **strip** command.

Syntax

►► -s ◀◀

Defaults

The symbol table, line number information, and relocation information are included in the output file.

Usage

Specifying **-s** saves space, but limits the usefulness of traditional debug programs when you are generating debug information using options such as **-g**.

Predefined macros

None.

Related information

- “-g” on page 108

-S

Category

Output control

Pragma equivalent

None.

Purpose

Generates an assembler language file for each source file.

The resulting file has an **.s** suffix and can be assembled to produce object **.o** files or an executable file (**a.out**).

Syntax

►► — -S ————— ◀◀

Defaults

Not applicable.

Usage

Note: You can not use this option for a program that targets both PPU and SPU and that is compiled using one of the cbe invocation commands, such as **cbexlc**.
invocation myprogram.s

will invoke the assembler, and if successful, the linker to create an executable file, a.out.

If you specify **-S** with **-E** or **-P**, **-E** or **-P** takes precedence. Order of precedence holds regardless of the order in which they were specified on the command line.

You can use the **-o** option to specify the name of the file produced only if no more than one source file is supplied. For example, the following is *not* valid:
invocation myprogram1.c myprogram2.c -o -S

Predefined macros

None.

Examples

To compile myprogram.c to produce an assembler language file myprogram.s, enter:
invocation myprogram.c -S

To assemble this program to produce an object file myprogram.o, enter:
invocation myprogram.s -c

To compile myprogram.c to produce an assembler language file asmprogram.s, enter:
invocation myprogram.c -S -o asmprogram.s

Related information

- “-E” on page 87
- “-P” on page 184

-qsaveopt

Category

Object code control

Pragma equivalent

None.

Purpose

Saves the command-line options used for compiling a source file, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.

Syntax

►► -q  ►►


Defaults

-qnosaveopt

Usage

This option has effect only when compiling to an object (.o) file (that is, using the -c option). Though each object may contain multiple compilation units, only one copy of the command-line options is saved. Compiler options specified with pragma directives are ignored.

Command-line compiler options information is copied as a string into the object file, using the following format:

►► @(#)—opt—  ►►

where:

c Signifies a C language compilation.

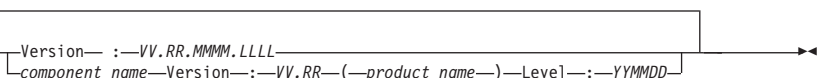
C Signifies a C++ language compilation.

invocation

Shows the command used for the compilation, for example, **ppuxlc**.

options The list of command line options specified on the command line, with individual options separated by spaces.

Compiler version and release information, as well as the version and level of each component invoked during compilation, are also saved to the object file in the format:

►► @(#)—version—  ►►

where:

V Represents the version.

R Represents the release.

M Represents the modification.

L Represents the level.

component_name

Specifies the components that were invoked for this compilation, such as the low-level optimizer.

product_name

Indicates the product to which the component belongs (for example, C/C++ or Fortran).

YYMMDD

Represents the year, month, and date of the installed update. If the update installed is at the base level, the level is displayed as BASE.

If you want to simply output this information to standard output without writing it to the object file, use the **-qversion** option.

Predefined macros

None.

Examples

Compile `t.c` with the following command:

```
invocation t.c -c -qsaveopt -qhot
```

Issuing the **strings -a** command on the resulting `t.o` object file produces information similar to the following:

```
opt c /opt/ibmcmp/vac/10.1/bin/xlc t.f -c -qsaveopt -qhot
version IBM XL C/C++ for Multicore Acceleration for Linux, V10.1
version Version: 10.01.0000.0000
version Driver Version: 10.01(C/C++) Level: YYMMDD
version Front End Version: 10.01(C/C++) Level: YYMMDD
version C Front End Version : 10.01(C/C++) Level: YYMMDD
version High Level Optimizer Version: 10.01(C) and 12.01(Fortran) Level: YYMMDD
version Low Level Optimizer Version: 10.01(C) and 12.01(Fortran) Level: YYMMDD
```

`c` identifies the source used as C, `/opt/ibmcmp/xlc/cbe/10.1/bin/ppuxlc` shows the invocation command used, and `-qhot -qsaveopt` shows the compilation options.

The remaining lines list each compiler component invoked during compilation, and its version and level. Components that are shared by multiple products may show more than one version number. Level numbers shown may change depending on the updates you have installed on your system.

Related information

- “-qversion” on page 258

-qshowinc

Category

Listings, messages, and compiler information

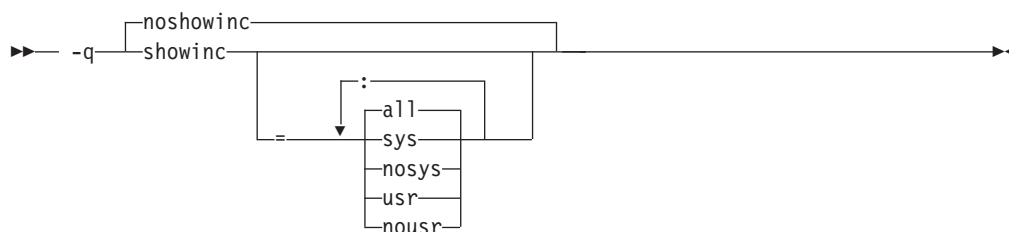
Pragma equivalent

```
#pragma options [no]showinc
```

Purpose

When used with **-qsource** option to generate a listing file, selectively shows user or system header files in the source section of the listing file.

Syntax



Defaults

-qnoshowinc: Header files included in source files are not shown in the source listing.

Parameters

all Shows both user and system include files in the program source listing.

sys

Shows system include files (that is, files included with the `#include <filename>` preprocessor directive) in the program source listing.

usr

Shows user include files (that is, files included with the `#include "filename"` preprocessor directive or with **-qinclude**) in the program source listing.

Specifying **showinc** with no suboptions is equivalent to **-qshowinc=sys : usr** and **-qshowinc=all**. Specifying **noshowinc** is equivalent to **-qshowinc=nosys : nousr**.

Usage

This option has effect only when the **-qlist** or **-qsource** compiler options is in effect.

Predefined macros

None.

Examples

To compile `myprogram.c` so that all included files appear in the source listing, enter:
invocation `myprogram.c -qsource -qshowinc`

Related information

- “-qsource” on page 218

-qshowmacros

Category

“Output control” on page 31

Pragma equivalent

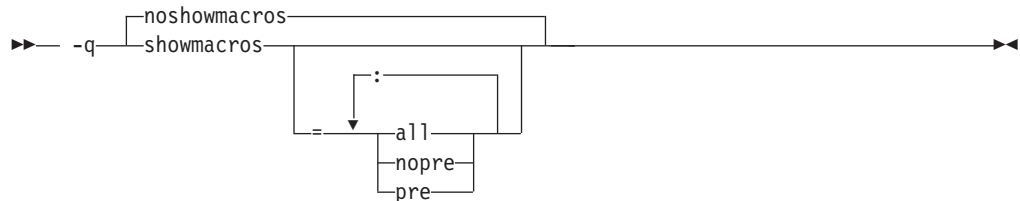
None

Purpose

Emits macro definitions to preprocessed output.

Emitting macros to preprocessed output can help determine functionality available in the compiler. The macro listing may prove useful for debugging complex macro expansions, as well.

Syntax



Defaults

-qnoshowmacros

Parameters

all Emits all macro definitions to preprocessed output. This is the same as specifying **-qshowmacros**.

pre | **nopre**

pre emits only predefined macro definitions to preprocessed output. **nopre** suppresses appending these definitions.

Usage

Note the following when using this option:

- This option has no effect unless preprocessed output is generated; for example, by using the **-E** or **-P** options.
- If a macro is defined and subsequently undefined before compilation ends, this macro will not be included in the preprocessed output.
- Only macros defined internally by the preprocessor are considered predefined; all other macros are considered as user-defined.

Related information

- “-E” on page 87
- “-P” on page 184

-qsmallstack

Category

Optimization and tuning

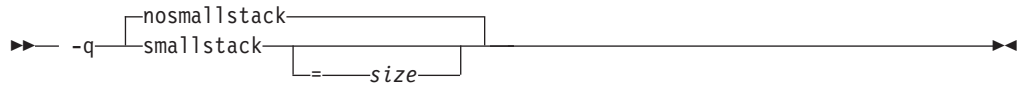
Pragma equivalent

None.

Purpose

Reduces the size of the stack frame.

Syntax



Parameters

size

Specifies the limit, in bytes, on variables declared in the parallel region, which might be accelerated to run on the SPU's to reside on the SPU stack, instead of the PPU heap. Note that the option has no effect when **-qnosmp** is in effect.

size can be an integer number between 0 and 262144 (256k). The default value will be determined by the compiler during tuning.

Defaults

-qnosmallstack

Usage

Programs that allocate large amounts of data to the stack, such as threaded programs, may result in stack overflows. This option can reduce the size of the stack frame to help avoid overflows.

This option is only valid when used together with IPA (**-qipa**, **-O4**, **-O5** compiler options).

Specifying this option may adversely affect program performance.

Predefined macros

None.

Examples

To compile `myprogram.c` to use a small stack frame, enter:

```
invocation myprogram.c -qipa -qsmallstack
```

Related information

- “-g” on page 108
- “-qipa” on page 132
- “-O, -qoptimize” on page 179

-qsmp

Category

Optimization and tuning

Pragma equivalent

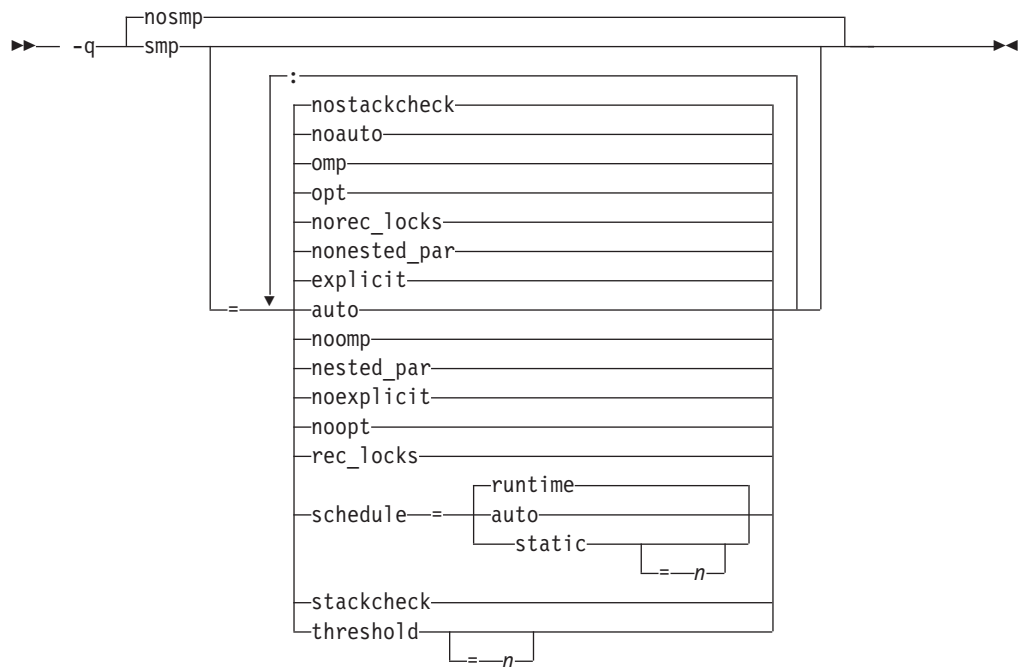
None.

Purpose

Enables parallelization of program code.

This option can only be used with the single-source compiler invocations, that is, one that is prefixed with **cbe**, such as **cbexlc** or **cbexlc++**.

Syntax



Defaults

`-qsmp=omp`

Parameters

auto | **noauto**

The **auto** suboption is ignored if specified. When **noauto** is in effect, only program code explicitly parallelized with OpenMP directives is optimized.

explicit | **noexplicit**

Enables or disables directives controlling explicit parallelization of loops.

nested_par | **nonested_par**

By default, the compiler serializes a nested parallel construct. When **nested_par** is in effect, the compiler parallelizes prescriptive nested parallel constructs.

This includes not only the loop constructs that are nested within a scoping unit but also parallel constructs in subprograms that are referenced (directly or indirectly) from within other parallel constructs. Note that this suboption has no effect on loops that are automatically parallelized. In this case, at most one loop in a loop nest (in a scoping unit) will be parallelized. **nested_par** does not

provide true nested parallelism because it does not cause new team of threads to be created for nested parallel regions. Instead, threads that are currently available are reused.

This suboption should be used with caution. Depending on the number of threads available and the amount of work in an outer loop, inner loops could be executed sequentially even if this option is in effect. Parallelization overhead may not necessarily be offset by program performance gains.

Note that the implementation of the **nested_par** suboption does not comply with the OpenMP API. If you specify this suboption, the runtime library uses the same threads for the nested constructs that it used for the enclosing constructs.

omp | noomp

Enforces or relaxes strict compliance to the OpenMP standard. When **omp** is in effect, **noauto** is implied and only OpenMP parallelization directives are recognized. The compiler issues warning messages if your code contains any language constructs that do not conform to the OpenMP API.

opt | noopt

Enables or disables optimization of parallelized program code. When **noopt** is in effect, the compiler will do the smallest amount of optimization that is required to parallelize the code. This is useful for debugging because **-qsmp** enables the **-O2** and **-qhot** options by default, which may result in the movement of some variables into registers that are inaccessible to the debugger. However, if the **-qsmp=noopt** and **-g** options are specified, these variables will remain visible to the debugger.

rec_locks | norec_locks

Determines whether recursive locks are used. When **rec_locks** is in effect, nested critical sections will not cause a deadlock. Note that the **rec_locks** suboption specifies behavior for critical constructs that is inconsistent with the OpenMP API.

schedule

Specifies the type of scheduling algorithms and, except in the case of **auto**, chunk size (n) that are used for loops to which no other scheduling algorithm has been explicitly assigned in the source code. Suboptions of the **schedule** suboption are as follows:

auto

With **auto**, scheduling is delegated to the compiler and runtime system. The compiler and runtime system can choose any possible mapping of iterations to threads (including all possible valid schedules) and these may be different in different loops. Chunk size (n) must not be specified when using **auto**. If chunk size (n) is specified, the compiler issues a severe error message. Note that when both the **-qsmp=schedule** option and **OMP_SCHEDULE** are used, the option will override the environment variable.

This option does not have effect on compilations with the single-source compiler invocations, such as **cbxlc**.

runtime

Specifies that the chunking algorithm will be determined at run time.

This option does not have effect on compilations with the single-source compiler invocations, such as **cbxlc**.

static[=*n*]

The iterations of a loop are divided into chunks containing *n* iterations each. Each thread is assigned chunks in a "round-robin" fashion. This is known as *block cyclic scheduling*. If the value of *n* is 1, then the scheduling type is specifically referred to as *cyclic scheduling*.

If *n* is not specified, the chunks will contain **ceiling**(*number_of_iterations* / *number_of_threads*) iterations. Each thread is assigned one of these chunks. This is known as *block scheduling*.

If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

n Must be an integer of value 1 or greater.

Specifying **schedule** with no suboption is equivalent to **schedule=runtime**.

stackcheck | nostackcheck

The **stackcheck** suboption is ignored if specified. Causes the compiler to check for stack overflow by worker threads at run time, and issue a warning if the remaining stack size is less than the number of bytes specified by the option.

threshold[=*n*]

When **-qsmp=auto** is in effect, controls the amount of automatic loop parallelization that occurs. The value of *n* represents the minimum amount of work required in a loop in order for it to be parallelized. Currently, the calculation of "work" is weighted heavily by the number of iterations in the loop. In general, the higher the value specified for *n*, the fewer loops are parallelized. Specifying a value of 0 instructs the compiler to parallelize all auto-parallelizable loops, whether or not it is profitable to do so. Specifying a value of 100 instructs the compiler to parallelize only those auto-parallelizable loops that it deems profitable. Specifying a value of greater than 100 will result in more loops being serialized.

n Must be a positive integer of 0 or greater.

If you specify **threshold** with no suboption, the program uses a default value of 100.

Specifying **-qsmp** without suboptions is equivalent to:

-qsmp=noauto:explicit:opt:omp:norec_locks:nonested_par:schedule=runtime:nostackcheck:threshold=100

Usage

- If you use the **-qsmp** option to compile any source file in a program, then you must specify the **-qsmp** option at link time as well, unless you link by using the **ppu-ld** or **spu-ld** command.
- Object files generated with the **-qsmp=opt** option can be linked with object files generated with **-qsmp=noopt**. The visibility within the debugger of the variables in each object file will not be affected by linking.
- The **-qnosmp** default option setting specifies that no code should be generated for parallelization directives, though syntax checking will still be performed. Use **-qignprag=omp** to completely ignore parallelization directives.
- Specifying **-qsmp** implicitly sets **-O2**. The **-qsmp** option overrides **-qnooptimize**, but does not override **-O3**, **-O4**, or **-O5**. When debugging parallelized program code, you can disable optimization in parallelized program code by specifying **qsmp=noopt**.

- The **-qsmp=noopt** suboption overrides performance optimization options anywhere on the command line unless **-qsmp** appears after **-qsmp=noopt**. For example, **-qsmp=noopt -O3** is equivalent to **-qsmp=noopt**, while **-qsmp=noopt -O3 -qsmp** is equivalent to **-qsmp -O3**.

Related information

- “-O, -qoptimize” on page 179
- “-qthreaded (PPU only)” on page 244
- “Environment variables for parallel processing” on page 23
- “Pragma directives for parallel processing” on page 312
- “Built-in functions for parallel processing” on page 366

-qsource

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]source

Purpose

Produces a compiler listing file that includes the source section of the listing and provides additional source information when printing error messages.

When **source** is in effect, a listing file is generated with a .lst suffix for each source file named on the command line. For details of the contents of the listing file, see “Compiler listings” on page 18.

Syntax

→ -q nosource
source →

Defaults

-qnosource

Usage

You can selectively print parts of the source by using pairs of **#pragma options source** and **#pragma options nosource** preprocessor directives throughout your source program. The source following **#pragma options source** and preceding **#pragma options nosource** is printed.

The **-qnoprint** option overrides this option.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce a compiler listing that includes the source code, enter:

```
invocation myprogram.c -qsource
```

Related information

- “-qlist” on page 164
- “-qlistopt” on page 165
- “-qprint” on page 193

-qsourcetype

Category

Input control

Pragma equivalent

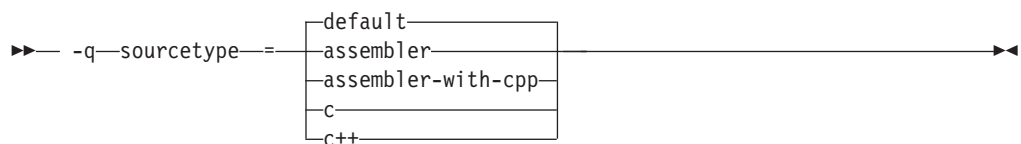
None.

Purpose

Instructs the compiler to treat all recognized source files as a specified source type, regardless of the actual file name suffix.

Ordinarily, the compiler uses the file name suffix of source files specified on the command line to determine the type of the source file. For example, a `.c` suffix normally implies C source code, and a `.C` suffix normally implies C++ source code. The **-qsourcetype** option instructs the compiler to not rely on the file name suffix, and to instead assume a source type as specified by the option.

Syntax



Defaults

`-qsourcetype=default`

Parameters

assembler

All source files following the option are compiled as if they are assembler language source files.

assembler-with-cpp

All source files following the option are compiled as if they are assembler language source files that need preprocessing.

- c** All source files following the option are compiled as if they are C language source files.

c++

All source files following the option are compiled as if they are C++ language source files. This suboption is equivalent to the `-+` option.

default

The programming language of a source file is implied by its file name suffix.

Usage

If you do not use this option, files must have a suffix of `.c` to be compiled as C files, and `.C` (uppercase C), `.cc`, `.cp`, `.cpp`, `.cxx`, or `.c++` to be compiled as C++ files.

This option applies whether the file system is case-sensitive or not. That is, even in a case-insensitive file system, where `file.c` and `file.C` refer to the same physical file, the compiler still recognizes the case difference of the file name argument on the command line and determines the source type accordingly.

Note that the option only affects files that are specified on the command line *following* the option, but not those that precede the option. Therefore, in the following example:

```
invocation goodbye.C -qsourcetype=c hello.C
```

`hello.C` is compiled as a C source file, but `goodbye.C` is compiled as a C++ file.

The `-qsourcetype` option should not be used together with the `-+` option.

Predefined macros

None.

Examples

To treat the source file `hello.C` as being a C language source file, enter:

```
invocation -qsourcetype=c hello.C
```

Related information

- “`-+` (plus sign) (C++ only)” on page 47

-qspill

Category

Compiler customization

Pragma equivalent

`#pragma options [no]spill`

Purpose

Specifies the size (in bytes) of the register spill space, the internal program storage areas used by the optimizer for register spills to storage.

Syntax

►► — `-qspill=size` —————►►

Defaults

`-qspill=512`

Parameters

size

An integer representing the number of bytes for the register allocation spill area.

Usage

If your program is very complex, or if there are too many computations to hold in registers at one time and your program needs temporary storage, you might need to increase this area. Do not enlarge the spill area unless the compiler issues a message requesting a larger spill area. In case of a conflict, the largest spill area specified is used.

Predefined macros

None.

Examples

If you received a warning message when compiling `myprogram.c` and want to compile it specifying a spill area of 900 entries, enter:

invocation `myprogram.c -qspill=900`

-qsrcmsg (C only)

Category

Listings, messages, and compiler information

Pragma equivalent

`#pragma options [no]srcmsg`

Purpose

Adds the corresponding source code lines to diagnostic messages generated by the compiler.

When **nosrcmsg** is in effect, the error message simply shows the file, line and column where the error occurred. When **srcmsg** is in effect, the compiler reconstructs the source line or partial source line to which the diagnostic message refers and displays it before the diagnostic message. A pointer to the column position of the error may also be displayed.

Syntax

►► -q  

Defaults

-qnosrcmsg

Usage

When **srcmsg** is in effect, the reconstructed source line represents the line as it appears after macro expansion. At times, the line may be only partially reconstructed. The characters "... " at the start or end of the displayed line indicate that some of the source line has not been displayed.

Use **-qnosrcmsg** to display concise messages that can be parsed.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the source line is displayed along with the diagnostic message when an error occurs, enter:

invocation `myprogram.c -qsrcmsg`

-qstaticinline (C++ only)

Category

Language element control

Pragma equivalent

None.

Purpose

Controls whether inline functions are treated as having `static` or `extern` linkage.

When **-qnostaticinline** is in effect, the compiler treats inline functions as `extern`: only one function body is generated for a function marked with the `inline` function specifier, regardless of how many definitions of the same function appear in different source files. When **-qstaticinline** is in effect, the compiler treats inline functions as having `static` linkage: a separate function body is generated for each definition in a different source file of the same function marked with the `inline` function specifier.

Syntax

►► -q  

Defaults

`-qnostaticinline`

Usage

When **-qnostaticinline** is in effect, any redundant functions definitions for which no bodies are generated are discarded by default; you can use the **-qkeepinlines** option to change this behavior.

Predefined macros

None.

Examples

Using the **-qstaticinline** option causes function `f` in the following declaration to be treated as static, even though it is not explicitly declared as such. A separate function body is created for each definition of the function. Note that this can lead to a substantial increase in code size.

```
inline void f() { /*...*/};
```

Related information

- "Linkage of inline functions" in the *XL C/C++ Language Reference*
- "`-qkeepinlines` (C++ only)" on page 143

-qstaticlink

Category

Linking

Pragma equivalent

None.

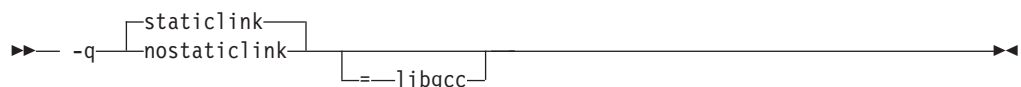
Purpose

Controls how shared and non-shared runtime libraries are linked into an application.

When **-qstaticlink** is in effect, the compiler links only static libraries with the object file named in the invocation. When **-qnostaticlink** is in effect, the compiler links shared libraries with the object file named in the invocation.

This option provides the ability to specify linking rules that are equivalent to those implied by the GNU options **-static**, **-static-libgcc**, and **-shared-libgcc**, used singly and in combination.

Syntax



Defaults

-qstaticlink

Parameters

libgcc

When specified together with **nostaticlink**, the compiler links to the shared version of libgcc. When specified together with **staticlink**, the compiler links to the static version of libgcc.

Usage

The following table shows the equivalent GNU and XL C/C++ options for specifying linkage of shared and non-shared libraries.

Table 22. Option mappings: control of the GNU linker

GNU option	Meaning	XL C/C++ option
-shared	Build a shared object.	-qmkshrobj
-static	Build a static object and prevent linking with shared libraries. Every library linked to must be a static library.	-qstaticlink
-shared-libgcc	Link with the shared version of libgcc.	-qnostaticlink=libgcc
-static-libgcc	Link with the static version of libgcc.	-qstaticlink=libgcc

WARNING: Any use of third-party libraries or products is subject to the provisions in their respective licenses. Using the **-qstaticlink** option can have significant legal consequences for the programs you compile. IBM strongly recommends that you seek legal advice before using this option.

Predefined macros

None.

-qstatsym

Category

Object code control

Pragma equivalent

None.

Purpose

Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the symbol table of the object file.

Syntax

►► — -q nostatsym
statsym ◀◀

Defaults

-qnostatsym: Static variables are not added to the symbol table. However, static functions are added to the symbol table.

Predefined macros

None.

Examples

To compile myprogram.c so that static symbols are added to the symbol table, enter:

invocation myprogram.c -qstatsym

-qstdinc

Category

Input control

Pragma equivalent

#pragma options [no]stdinc

Purpose

Specifies whether the standard include directories are included in the search paths for system and user header files.

When **-qstdinc** is in effect, the compiler searches the following directories for header files:

- The directory specified in the configuration file for the XL C and C++ header files (this is normally /opt/ibmcmp/xlc/cbe/10.1/include/) or by the **-qc_stdinc** and **-qcpp_stdinc** options
- The directory specified in the configuration file for the system header files or by the **-qgcc_c_stdinc** and **-qgcc_cpp_stdinc** options

When **-qnostdinc** is in effect, these directories are excluded from the search paths. The only directories to be searched are:

- directories in which source files containing #include "filename" directives are located
- directories specified by the **-I** option
- directories specified by the **-qinclude** option

Syntax

►► -q stdinc
nostdinc ◀◀

Defaults

-qstdinc

Usage

The search order of header files is described in “Directory search sequence for include files” on page 12.

This option only affects search paths for header files included with a relative name; if a full (absolute) path name is specified, this option has no effect on that path name.

The last valid pragma directive remains in effect until replaced by a subsequent pragma.

Predefined macros

None.

Examples

To compile `myprogram.c` so that *only* the directory `/tmp/myfiles` (in addition to the directory containing `myprogram.c`) is searched for the file included with the `#include "myinc.h"` directive, enter:

invocation `myprogram.c -qnostdinc -I/tmp/myfiles`

Related information

- “`-qc_stdinc` (C only)” on page 78
- “`-qcpp_stdinc` (C++ only)” on page 79
- “`-qgcc_c_stdinc` (C only)” on page 109
- “`-qgcc_cpp_stdinc` (C++ only)” on page 110
- “`-I`” on page 118
- “Directory search sequence for include files” on page 12

-qstdmain (SPU only)

Category

Linking

Pragma equivalent

None.

Purpose

Links your SPU program with the startup routines required to use C99-style main function arguments.

Syntax

►► — `-qstdmain` ————— ◀◀

Defaults

By default, the output object is linked with the runtime libraries and startup routines that use an SPU style main function argument list.

Usage

This option is only valid with the SPU invocation commands.

Predefined macros

None.

-qstrict

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]strict

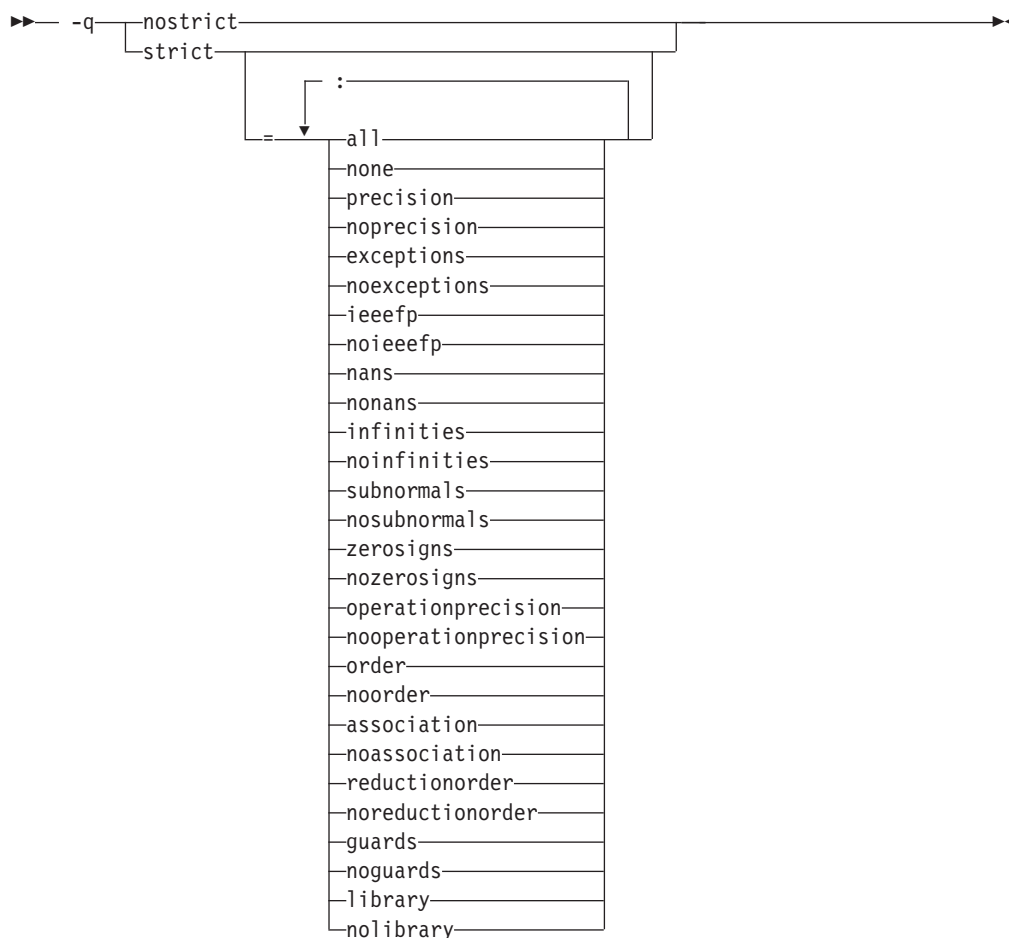
#pragma option_override (*function_name*, "opt (*suboption_list*)")

Purpose

Ensures that optimizations done by default at optimization levels **-O3** and higher, and, optionally at **-O2**, do not alter the semantics of a program.

This option is intended for situations where the changes in program execution in optimized programs produce different results from unoptimized programs.

Syntax



Defaults

- Always **-qstrict** or **-qstrict=all** when the **-qnoot** or **-O0** optimization level is in effect
- **-qstrict** or **-qstrict=all** is the default when the **-O2** or **-O** optimization level is in effect
- **-qnostrict** or **-qstrict=none** is the default when **-O3** or a higher optimization level is in effect

Parameters

The **-qstrict** suboptions include the following:

all | **none**

all disables all semantics-changing transformations, including those controlled by the **ieeefp**, **order**, **library**, **precision**, and **exceptions** suboptions. **none** enables these transformations.

precision | **noprecision**

precision disables all transformations that are likely to affect floating-point precision, including those controlled by the **subnormals**, **operationprecision**, **association**, **reductionorder**, and **library** suboptions. **noprecision** enables these transformations.

exceptions | **noexceptions**

exceptions disables all transformations likely to affect exceptions or be affected

by them, including those controlled by the **nans**, **infinities**, **subnormals**, **guards**, and **library** suboptions. **noexceptions** enables these transformations.

ieee^{ie}fp | noieee^{ie}fp (PPU only)

ieee^{ie}fp disables transformations that affect IEEE floating-point compliance, including those controlled by the **nans**, **infinities**, **subnormals**, **zerosigns**, and **operationprecision** suboptions. **noieee^{ie}fp** enables these transformations.

nans | nonans

nans disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce IEEE floating-point NaN (not-a-number) values. **nonans** enables these transformations.

infinities | noinfinities

infinities disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce floating-point infinities. **noinfinities** enables these transformations.

subnormals | nosubnormals

subnormals disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce IEEE floating-point subnormals (formerly known as denorms). **nosubnormals** enables these transformations.

zerosigns | nozerosigns

zerosigns disables transformations that may affect or be affected by whether the sign of a floating-point zero is correct. **nozerosigns** enables these transformations.

operationprecision | nooperationprecision

operationprecision disables transformations that produce approximate results for individual floating-point operations. **nooperationprecision** enables these transformations.

order | noorder

order disables all code reordering between multiple operations that may affect results or exceptions, including those controlled by the **association**, **reductionorder**, and **guards** suboptions. **noorder** enables code reordering.

association | noassociation

association disables reordering operations within an expression. **noassociation** enables reordering operations.

reductionorder | noreductionorder

reductionorder disables parallelizing floating-point reductions. **noreductionorder** enables parallelizing these reductions.

guards | noguards

guards disables moving operations past guards (that is, past **IF** statements or out of loops) or calls which control whether the operation should be executed or not. **noguards** enables moving these operations.

library | nolibrary

library disables transformations that affect floating-point library functions; for example, transformations that replace floating-point library functions with other library functions or with constants. **nolibrary** enables these transformations.

Usage

The **all**, **precision**, **exceptions**, **ieee^{ie}fp**, and **order** suboptions and their negative forms are group suboptions that affect multiple, individual suboptions. For many

situations, the group suboptions will give sufficient granular control over transformations. Group suboptions act as if either the positive or the no form of every suboption of the group is specified. Where necessary, individual suboptions within a group (like **subnormals** or **operationprecision** within the **precision** group) provide control of specific transformations within that group.

With **-qnostrict** or **-qstrict=none** in effect, the following optimizations are turned on:

- Code that may cause an exception may be rearranged. The corresponding exception might happen at a different point in execution or might not occur at all. (The compiler still tries to minimize such situations.)
- Floating-point operations may not preserve the sign of a zero value. (To make certain that this sign is preserved, you also need to specify **-qfloat=rrm**, **-qfloat=nomaf**, or **-qfloat=strictnmaf**.)
- Floating-point expressions may be reassociated. For example, $(2.0*3.1)*4.2$ might become $2.0*(3.1*4.2)$ if that is faster, even though the result might not be identical.
- The **fltint** and **rsqrt** suboptions of the **-qfloat** option are turned on. You can turn them off again by also using the **-qstrict** option or the **nofltint** and **norsqrt** suboptions of **-qfloat**. With lower-level or no optimization specified, these suboptions are turned off by default.

Specifying various **-qstrict[=suboptions]** or **-qnostrict** combinations sets **-qfloat** suboptions as follows:

- **-qstrict** or **-qstrict=all** sets **-qfloat=norsqrt:rngchk**. **-qnostrict** or **-qstrict=none** sets **-qfloat=rsqrt:norngchk**.
- **-qstrict=operationprecision** or **-qstrict=exceptions** sets **-qfloat=nofltint**. Specifying both **-qstrict=nooperationprecision** and **-qstrict=noexceptions** sets **-qfloat=fltint**.
- **-qstrict=infinities**, **-qstrict=operationprecision**, or **-qstrict=exceptions** sets **-qfloat=norsqrt**.
- **-qstrict=noinfinities:nooperationprecision:noexceptions** sets **-qfloat=rsqrt**.
- **-qstrict=nans**, **-qstrict=infinities**, **-qstrict=zerosigns**, or **-qstrict=exceptions** sets **-qfloat=rngchk**. Specifying all of **-qstrict=nonans:nozerosigns:noexceptions** or **-qstrict=noinfinities:nozerosigns:noexceptions**, or any group suboptions that imply all of them, sets **-qfloat=norngchk**.

To override any of these settings, specify the appropriate **-qfloat** suboptions after **-q[no]strict[=suboption_list]** on the command line.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the aggressive optimizations of **-O3** are turned off, and division by the result of a square root is replaced by multiplying by the reciprocal (**-qfloat=rsqrt**), enter:

```
invocation myprogram.c -O3 -qstrict -qfloat=rsqrt
```

To enable all transformations except those affecting precision, specify:

```
invocation myprogram.c -qstrict=none:precision
```

To disable all transformations except those involving NaNs and infinities, specify:
invocation `myprogram.c -qstrict=all:nonans:notinfinities`

Related information

- “-qfloat” on page 98
- “-qhot” on page 115
- “-O, -qoptimize” on page 179

-qstrict_induction

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.

Syntax



Defaults

- **-qstrict_induction**
- **-qnostrict_induction** when **-O2** or higher optimization level is in effect

Usage

When using **-O2** or higher optimization, you can specify **-qstrict_induction** to prevent optimizations that change the result of a program if truncation or sign extension of a loop induction variable should occur as a result of variable overflow or wrap-around. However, use of **-qstrict_induction** is generally not recommended because it can cause considerable performance degradation.

Predefined macros

None.

Related information

- “-O, -qoptimize” on page 179

-qsuppress

Category

Listings, messages, and compiler information

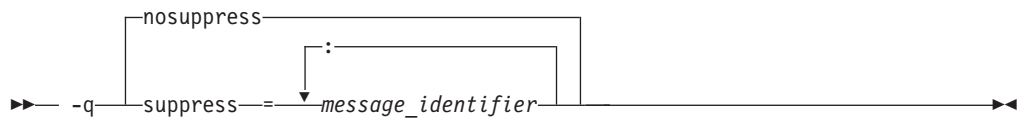
Pragma equivalent

"#pragma report (C++ only)" on page 304

Purpose

Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.

Syntax



Defaults

-qnosuppress: All informational and warning messages are reported, unless set otherwise with the **-qflag** option.

Parameters

message_identifier

Represents a message identifier. The message identifier must be in the following format:

15dd-number

where:

dd Is the two-digit code representing the compiler component that produces the message. See "Compiler message format" on page 15 for descriptions of these.

number

Is the message number.

Usage

You can only suppress information (I) and warning (W) messages. You cannot suppress other types of messages, such as (S) and (U) level messages. Note that informational and warning messages that supply additional information to a severe error cannot be disabled by this option.

To suppress all informational and warning messages, you can use the **-w** option.

To suppress IPA messages, enter **-qsuppress** before **-qipa** on the command line.

The **-qnosuppress** compiler option cancels previous settings of **-qsuppress**.

Predefined macros

None.

Examples

If your program normally results in the following output:

```
"myprogram.c", line 1.1:1506-224 (I) Incorrect #pragma ignored
```

you can suppress the message by compiling with:

```
invocation myprogram.c -qsuppress=1506-224
```

Related information

- “-qflag” on page 97

-qswcache_size (SPU only)

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Sets the size of the software cache used in effective address space support.

Syntax

►► — -q—swcache_size—=size— ◀◀

Defaults

The default size of the software cache is 64 KB.

Parameters

size

The size of the software cache in KB; it can be 8, 16, 32, 64, or 128.

Usage

This is a link-time option that is supported on the SPU using one of the `spu` invocation commands, such as **spuxlc** or when compiling a program that targets both PPU and SPU using one of the `cbe` invocation commands, such as **cbexlc**.

You may need to experiment with different cache sizes to determine the optimal cache size for your SPU application. The optimal size will likely depend on the number of PPU references in your application and how much of the SPU's memory you are willing to devote to the software cache.

Predefined macros

None.

Examples

To compile the program `myprogram.c` with the software cache size set to 16 KB, enter:

```
spuxlc myprogram.c -qswcache_size=16
```

Related information

- "Accessing PPU variables from SPU programs" in the *XL C/C++ Optimization and Programming Guide*
- "The `__ea` type qualifier" in the *XL C/C++ Language Reference*

-qsymtab (C only)

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Determines the information that appears in the symbol table.

Syntax

►► -q-symtab=[unref|static] ◀◀

Defaults

Static variables and unreferenced typedef, structure, union, and enumeration declarations are not included in the symbol table of the object file.

Parameters

unref

When used with the `-g` option, specifies that debugging information is included for unreferenced typedef declarations, struct, union, and enum type definitions in the symbol table of the object file. This suboption is equivalent to `-qdbxextra`.

Using `-qsymtab=unref` may make your object and executable files larger.

static

Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the symbol table of the object file. This suboption is equivalent to `-qstatsym`.

Predefined macros

None.

Examples

To compile `myprogram.c` so that static symbols are added to the symbol table, enter:
invocation `myprogram.c -qsyntab=static`

To compile `myprogram.c` so that unreferenced typedef, structure, union, and enumeration declarations are included in the symbol table for use with a debugger, enter:

invocation `myprogram.c -g -qsyntab=unref`

Related information

- “-g” on page 108
- “-qdbxextra (C only)” on page 82
- “-qstatsym” on page 224

-qsyntaxonly (C only)

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Performs syntax checking without generating an object file.

Syntax

►► — -q—syntaxonly ————— ◀◀

Defaults

By default, source files are compiled and linked to generate an executable file.

Usage

The **-P**, **-E**, and **-C** options override the **-qsyntaxonly** option, which in turn overrides the **-c** and **-o** options.

The **-qsyntaxonly** option suppresses only the generation of an object file. All other files, such as listing files, are still produced if their corresponding options are set.

Predefined macros

None.

Examples

To check the syntax of `myprogram.c` without generating an object file, enter:
invocation `myprogram.c -qsyntaxonly`

Related information

- “-C, -C!” on page 66
- “-c” on page 65
- “-E” on page 87
- “-o” on page 178
- “-P” on page 184

-t

Category

Compiler customization

Pragma equivalent

None.

Purpose

Applies the prefix specified by the **-B** option to the designated components.

Syntax




Defaults

The default paths for all of the compiler executables are defined in the compiler configuration file.

Parameters

The following table shows the correspondence between **-t** parameters and the component executable names:

Parameter	Description	Executable name
a	Assembler	ppu-as or spu-as
b	Low-level optimizer	xlCcode
c	Compiler front end	xlcentry, xlCentry
 C	C++ compiler front end	xlCentry
d	Disassembler	dis
I	High-level optimizer, compile step	ipa
L	High-level optimizer, link step	ipa
l	Linker	ppu-ld or spu-ld

Usage

This option is intended to be used together with the **-B***prefix* option. If **-B** is specified without the *prefix*, the default prefix is `/lib/o`. If **-B** is not specified at all, the prefix of the standard program names is `/lib/n`.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the name `/u/newones/compilers/` is prefixed to the compiler and assembler program names, enter:

```
invocation myprogram.c -B/u/newones/compilers/ -tca
```

Related information

- “-B” on page 63

-qtabsize

Category

Language element control

Pragma equivalent

`#pragma options tabsize`

Purpose

Sets the default tab length, for the purposes of reporting the column number in error messages.

Syntax

►► — `-q—tabsize=—number—` —————►◄

Defaults

`-qtabsize=8`

Parameters

number

The number of character spaces representing a tab in your source program.

Usage

This option only affects error messages that specify the column number at which an error occurred.

Predefined macros

None.

Examples

To compile `myprogram.c` so the compiler considers tabs as having a width of one character, enter:

```
invocation myprogram.c -qtabsize=1
```

In this case, you can consider one character position (where each character and each tab equals one position, regardless of tab length) as being equivalent to one character column.

-qtbtable (PPU only)

Category

Object code control

Pragma equivalent

#pragma options ttable

Purpose

Controls the amount of debugging traceback information that is included in the object files.

Many performance measurement tools require a full traceback table to properly analyze optimized code. If a traceback table is generated, it is placed in the text segment at the end of the object code, and contains information about each function, including the type of function, as well as stack frame and register information.

Syntax

►► -q—tbtable—=

full
none
small

 ◀◀

Defaults

- `-qtbtable=full`
- `-qtbtable=small` when `-O` or higher optimization is in effect

Parameters

full

A full traceback table is generated, complete with name and parameter information.

none

No traceback table is generated. The stack frame cannot be unwound so exception handling is disabled.

small

The traceback table generated has no name or parameter information, but otherwise has full traceback capability. This suboption reduces the size of the program code.

Usage

This option applies only to 64-bit compilations, and is ignored if specified for a 32-bit compilation.

The **#pragma** options directive must be specified before the first statement in the compilation unit.

Predefined macros

None.

Related information

- “-g” on page 108

-qtempinc (C++ only)

Category

Template control

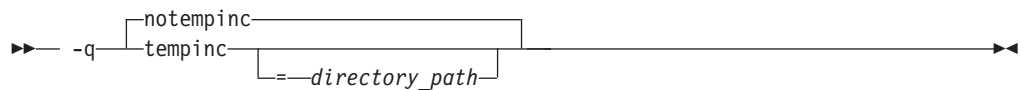
Pragma equivalent

None.

Purpose

Generates separate template include files for template functions and class declarations, and places these files in a directory which can be optionally specified.

Syntax



Defaults

-qnotempinc

Parameters

directory_path

The directory in which the generated template include files are to be placed.

Usage

The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive. Specifying **-qtempinc** implies **-qnotemplateregistry**. Similarly, specifying **-qtemplateregistry** implies **-qnotempinc**. However, specifying **-qnotempinc** does not imply **-qtemplateregistry**.

Specifying either **-qtempinc** or **-qtemplateregistry** implies **-qtmplinst=auto**.

Predefined macros

`__TEMPINC__` is predefined to 1 when **-qtempinc** is in effect; otherwise, it is not defined.

Examples

To compile the file `myprogram.C` and place the generated include files for the template functions in the `/tmp/mytemplates` directory, enter:

```
ppuxlc++ myprogram.C -qtempinc=/tmp/mytemplates
```

Related information

- “`#pragma` implementation (C++ only)” on page 287
- “`-qtmplinst` (C++ only)” on page 247
- “`-qtemplateregistry` (C++ only)” on page 242
- “`-qtemplaterecompile` (C++ only)” on page 241
- “Using C++ templates” in the *XL C/C++ Optimization and Programming Guide*.

-qtemplatedepth (C++ only)

Category

Template control

Pragma equivalent

None.

Purpose

Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.

Syntax

►► `-qtemplatedepth==number` ◀◀

Defaults

`-qtemplatedepth=300`

Parameters

number

The maximum number of recursive template instantiations. The number can be a value between 1 and `INT_MAX`. If your code attempts to recursively instantiate more templates than *number*, compilation halts and an error message is issued. If you specify an invalid value, the default value of 300 is used.

Usage

Note that setting this option to a high value can potentially cause an out-of-memory error due to the complexity and amount of code generated.

Predefined macros

None.

Examples

To allow the following code in `myprogram.cpp` to be compiled successfully:

```
template <int n> void foo() {  
    foo<n-1>();  
}  
  
template <> void foo<0>() {}  
  
int main() {  
    foo<400>();  
}
```

Enter:

```
ppuxlc++ myprogram.cpp -qtemplatedepth=400
```

Related information

- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qtemplaterecompile (C++ only)

Category

Template control

Pragma equivalent

None.

Purpose

Helps manage dependencies between compilation units that have been compiled using the **-qtemplateregistry** compiler option.

Syntax

►► — -q —  —►►

Defaults

-qtemplaterecompile

Usage

If a source file that has been compiled previously is compiled again, the **-qtemplaterecompile** option consults the template registry to determine whether changes to this source file require the recompile of other compilation units. This can occur when the source file has changed in such a way that it no longer references a given instantiation and the corresponding object file previously contained the instantiation. If so, affected compilation units will be recompiled automatically.

The **-qtemplaterecompile** option requires that object files generated by the compiler remain in the subdirectory to which they were originally written. If your automated build process moves object files from their original subdirectory, use the **-qnotemplaterecompile** option whenever **-qtemplateregistry** is enabled.

Predefined macros

None.

Related information

- “-qtmplinst (C++ only)” on page 247
- “-qtempinc (C++ only)” on page 239
- “-qtemplateregistry (C++ only)”
- “Using C++ templates” in the *XL C/C++ Optimization and Programming Guide*.

-qtemplateregistry (C++ only)

Category

Template control

Pragma equivalent

None.

Purpose

Maintains records of all templates as they are encountered in the source and ensures that only one instantiation of each template is made.

The first time that the compiler encounters a reference to a template instantiation, that instantiation is generated and the related object code is placed in the current object file. Any further references to identical instantiations of the same template in different compilation units are recorded but the redundant instantiations are not generated. No special file organization is required to use the **-qtemplateregistry** option.

Syntax

```
➡ -q notemplateregistry
    templateregistry -=file_path ⬅
```

Defaults

-qnotemplateregistry

Parameters

file_path

The path for the file that will contain the template instantiation information. If you do not specify a location the compiler saves all template registry information to the file `templateregistry` stored in the current working directory.

Usage

Template registry files must not be shared between different programs. If there are two or more programs whose source is in the same directory, relying on the default template registry file stored in the current working directory may lead to incorrect results.

The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive. Specifying **-qtempinc** implies **-qnotemplateregistry**. Similarly, specifying **-qtemplateregistry** implies **-qnotempinc**. However, specifying **-qtemplateregistry** does not imply **-qtempinc**.

Specifying either **-qtempinc** or **-qtemplateregistry** implies **-qtmplinst=auto**.

Predefined macros

None.

Examples

To compile the file `myprogram.C` and place the template registry information into the `/tmp/mytemplateregistry` file, enter:

```
ppuxlc++ myprogram.C -qtemplateregistry=/tmp/mytemplateregistry
```

Related information

- “-qtmplinst (C++ only)” on page 247
- “-qtempinc (C++ only)” on page 239
- “-qtemplaterecompile (C++ only)” on page 241
- “Using C++ templates” in the *XL C/C++ Optimization and Programming Guide*.

-qtempmax (C++ only)

Category

Template control

Pragma equivalent

None.

Purpose

Specifies the maximum number of template include files to be generated by the **-qtempinc** option for each header file.

Syntax

►► `-qtempmax=number` ◀◀

Defaults

`-qtempmax=1`

Parameters

number

The maximum number of template include files. The number can be a value between 1 and 99 999.

Usage

This option should be used when the size of files generated by the **-qtempinc** option become very large and take a significant amount of time to recompile when a new instance is created.

Instantiations are spread among the template include files.

Predefined macros

None.

Related information

- “-qtempinc (C++ only)” on page 239
- “Using C++ templates” in the *XL C/C++ Optimization and Programming Guide*.

-qthreaded (PPU only)

Category

Object code control

Pragma equivalent

None.

Purpose

Indicates to the compiler whether it must generate threadsafe code.

Always use this option when compiling or linking multithreaded applications. This option does not make code threadsafe, but it will ensure that code already threadsafe will remain so after compilation and linking. It also ensures that all optimizations are threadsafe.

Syntax

►► — -q —

nothreaded
threaded

 —►►

Defaults

- **-qnothreaded** for all invocation commands except those with the **_r** suffix
- **-qthreaded** for all **_r**-suffixed invocation commands

Usage

This option applies to both compile and linker operations.

To maintain thread safety, a file compiled with the **-qthreaded** option, whether explicitly by option selection or implicitly by choice of **_r** compiler invocation mode, must also be linked with the **-qthreaded** option.

Predefined macros

None.

-qtimestamps

Category

“Output control” on page 31

Pragma equivalent

none.

Purpose

Controls whether or not implicit time stamps are inserted into an object file.

Syntax

→ -q ————— →
 ↑ timestamps
 ↓ notimestamps

Defaults

-qtimestamps

Usage

By default, the compiler inserts an implicit time stamp in an object file when it is created. In some cases, comparison tools may not process the information in such binaries properly. Controlling time stamp generation provides a way of avoiding such problems. To omit the time stamp, use the option **-qnotimestamps**.

This option does not affect time stamps inserted by pragmas and other explicit mechanisms.

-qtls (PPU only)

Category

Object code control

Pragma equivalent

None.

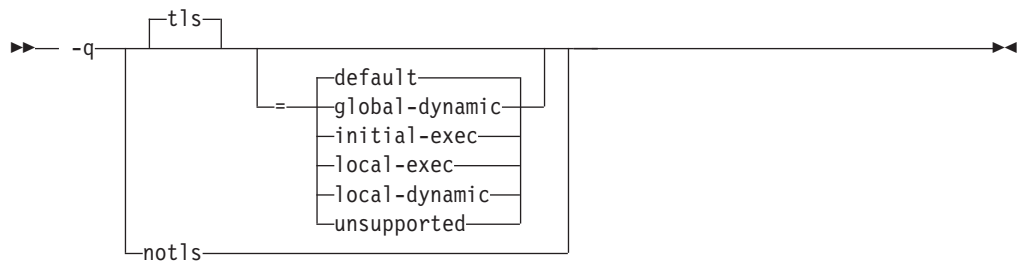
Purpose

Enables recognition of the **__thread** storage class specifier, which designates variables that are to be allocated threadlocal storage; and specifies the threadlocal storage model to be used.

When this option is in effect, any variables marked with the `__thread` storage class specifier are treated as local to each thread in a multi-threaded application. At run time, a copy of the variable is created for each thread that accesses it, and destroyed when the thread terminates. Like other high-level constructs that you can use to parallelize your applications, thread-local storage prevents race conditions to global data, without the need for low-level synchronization of threads.

Suboptions allow you to specify thread-local storage models, which provide better performance but are more restrictive in their applicability.

Syntax



Defaults

`-qtls=default`

Parameters

unsupported

The `__thread` keyword is not recognized and thread-local storage is not enabled. This suboption is equivalent to `-qnotls`.

global-dynamic

This model is the most general, and can be used for all thread-local variables.

initial-exec

This model provides better performance than the global-dynamic or local-dynamic models, and can be used for thread-local variables defined in dynamically-loaded modules, provided that those modules are loaded at the same time as the executable. That is, it can only be used when all thread-local variables are defined in modules that are not loaded through `dlopen`.

local-dynamic

This model provides better performance than the global-dynamic model, and can be used for thread-local variables defined in dynamically-loaded modules. However, it can only be used when all references to thread-local variables are contained in the same module in which the variables are defined.

local-exec

This model provides the best performance of all of the models, but can only be used when all thread-local variables are defined and referenced by the main executable.

default

Uses the appropriate model depending on the setting of the `-qpic` compiler option, which determines whether position-independent code is generated or not. When `-qpic` is in effect, this suboption results in `-qtls=global-dynamic`.

When **-qnopic** is in effect, this suboption results in **-qtls=initial-exec** (**-qpik** is in effect by default in 64-bit mode, and cannot be disabled).

Specifying **-qtls** with no suboption is equivalent to **-qtls=default**.

Predefined macros

None.

Related information

- “-qpik” on page 190
- “The `__thread` storage class specifier” in the *XL C/C++ Language Reference*

-qtmplinst (C++ only)

Category

Template control

Pragma equivalent

None.

Purpose

Manages the implicit instantiation of templates.

Syntax



Defaults

-qtmplinst=auto

Parameters

always

Instructs the compiler to always perform implicit instantiation. If specified, **-qtempinc** and **-qtemplateregistry** compiler options are ignored.

auto

Manages the implicit instantiations according to the **-qtempinc** and **-qtemplateregistry** options. If both **-qtempinc** and **-qtemplateregistry** are disabled, implicit instantiation will always be performed; otherwise if one of the options is enabled, the compiler manages the implicit instantiation according to that option.

noinline

Instructs the compiler to not perform any implicit instantiations. If specified, the **-qtempinc** and **-qtemplateregistry** compiler options are ignored.

none

Instructs the compiler to instantiate only inline functions. No other implicit instantiation is performed. If specified, **-qtempinc** and **-qtemplateregistry** compiler options are ignored.

Usage

You can also use **#pragma do_not_instantiate** to suppress implicit instantiation of selected template classes. See “**#pragma do_not_instantiate (C++ only)**” on page 281.

Predefined macros

None.

Related information

- “**-qtemplateregistry (C++ only)**” on page 242
- “**-qtempinc (C++ only)**” on page 239
- “**#pragma do_not_instantiate (C++ only)**” on page 281
- “Explicit instantiation” in the *XL C/C++ Language Reference*

-qtmplparse (C++ only)

Category

Template control

Pragma equivalent

None.

Purpose

Controls whether parsing and semantic checking are applied to template definitions.

Syntax



Defaults

-qtmplparse=no

Parameters

error

Treats problems in template definitions as errors, even if the template is not instantiated.

no Do not parse template definitions. This reduces the number of errors issued in code written for previous versions of VisualAge® C++ and predecessor products.

warn

Parses template definitions and issues warning messages for semantic errors.

Usage

This option applies to template definitions, not their instantiations. Regardless of the setting of this option, error messages are produced for problems that appear outside definitions. For example, messages are always produced for errors found during the parsing or semantic checking of constructs such as the following:

- return type of a function template
- parameter list of a function template

Predefined macros

None.

Related information

- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qtocdata (PPU only)

See "-qdataimported, -qdatalocal, -qtocdata (PPU only)" on page 81.

-qtrigraph

Category

Language element control

Pragma equivalent

None.

Purpose

Enables the recognition of trigraph key combinations to represent characters not found on some keyboards.

Syntax



Defaults

-qtrigraph

Usage

A trigraph is a combination of three-key character combinations that let you produce a character that is not available on all keyboards. For details, see "Trigraph sequences" in the *XL C/C++ Language Reference*.

▶ **C++** To override the default **-qtrigraph** setting, you must specify **-qnotrigraph** after the **-qlanglvl** option on the command line.

Predefined macros

None.

Related information

- "Trigraph sequences" in the *XL C/C++ Language Reference*
- "-qdigraph" on page 83
- "-qlanglvl" on page 148

-qtune

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.

Syntax

-qtune syntax



Defaults

- **-qtune=cellppu** for compilation targeting the PPU when using one of the ppu invocation commands, such as **ppuxlc**.
- **-qtune=edp** for compilation targeting the SPU when using one of the spu invocation commands, such as **spuxlc**.
- **-qtune=celledp** for compilation targeting both the PPU and SPU when using one of the cbe invocation commands, such as **cbexlc**.

Parameters

auto

Optimizations are tuned for the platform on which the application is compiled.

cell

Optimizations are tuned for the PPU and SPU when a program is compiled using one of the cbe invocation commands, such as **cbexlc**.

celledp

Optimizations are tuned for the PPU and SPU with enhanced double precision when a program is compiled using one of the cbe invocation commands, such as **cbexlc**.

cellppu (PPU only)

Optimizations are tuned for the PPU when a program is compiled using one of the ppu invocation commands, such as **ppuxlc**.

cellspu (SPU only)

Optimizations are tuned for the SPU when a program is compiled using one of the spu invocation commands, such as **spuxlc**.

edp (SPU only)

Optimizations are tuned for the enhanced CBEA-compliant processor with a double precision SPU when a program is compiled using one of the spu invocation commands, such as **spuxlc**.

Usage

Acceptable combinations of **-qarch** and **-qtune** are shown in the following table.

Table 23. Acceptable **-qarch/-qtune** combinations

-qarch option	Default -qtune setting	Available -qtune settings
cell	celledp	auto cell celledp
celledp	celledp	auto celledp
cellppu	cellppu	auto cellppu
cellspu	edp	auto cellspu edp
edp	edp	auto edp

Predefined macros

None.

Related information

- “-qarch” on page 57
- “-q32, -q64 (PPU only)” on page 49
- “Optimizing your applications” in the *XL C/C++ Optimization and Programming Guide*

-U

Category

Language element control

Pragma equivalent

None.

Purpose

Undefines a macro defined by the compiler or by the **-D** compiler option.

Syntax

►► — **-U**—*name*—————►►

Defaults

Many macros are predefined by the compiler; see Chapter 5, “Compiler predefined macros,” on page 325 for those that can be undefined (that is, are not *protected*). The compiler configuration file also uses the **-D** option to predefine several macro names for specific invocation commands; for details, see the configuration file for your system.

Parameters

name

The macro you want to undefine.

Usage

The **-U** option is *not* equivalent to the `#undef` preprocessor directive. It *cannot* undefine names defined in the source by the `#define` preprocessor directive. It can only undefine names defined by the compiler or by the **-D** option.

The **-Uname** option has a higher precedence than the **-Dname** option.

Predefined macros

None.

Examples

Assume that your operating system defines the name `__unix`, but you do not want your compilation to enter code segments conditional on that name being defined, compile `myprogram.c` so that the definition of the name `__unix` is nullified by entering:

```
invocation myprogram.c -U__unix
```

Related information

- “**-D**” on page 80

-qunroll

Category

Optimization and tuning

Pragma equivalent

`#pragma options [no]unroll`, `#pragma unroll`

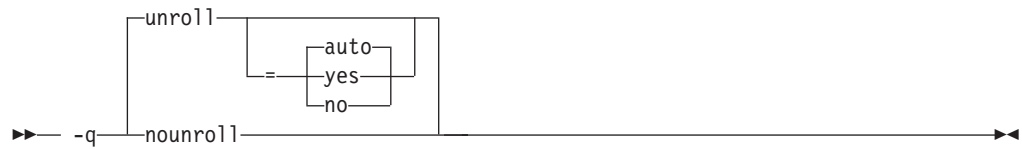
Purpose

Controls loop unrolling, for improved performance.

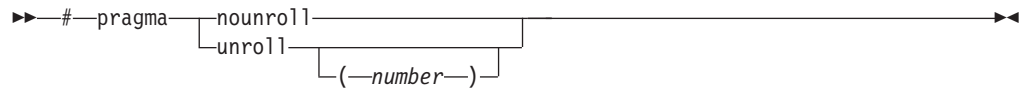
When **unroll** is in effect, the optimizer determines and applies the best unrolling factor for each loop; in some cases, the loop control may be modified to avoid unnecessary branching. The compiler remains the final arbiter of whether the loop is actually unrolled. You can use the **#pragma unroll** directive to gain more control over unrolling.

Syntax

Option syntax



Pragma syntax



Defaults

`-qunroll=auto`

Parameters

auto (option only)

Instructs the compiler to perform basic loop unrolling.

yes (option only)

Instructs the compiler to search for more opportunities for loop unrolling than that performed with **auto**. In general, this suboption has more chances to increase compile time or program size than **auto** processing, but it may also improve your application's performance.

no (option only)

Instructs the compiler to not unroll loops.

number (pragma only)

Forces *number* - 1 replications of the designated loop body or full unrolling of the loop, whichever occurs first. The value of *number* is unbounded and must be a positive integer. Specifying **#pragma unroll(1)** effectively disables loop unrolling, and is equivalent to specifying **#pragma nounroll**. If *number* is not specified and if **-qhot** or **-O4** or higher is specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Specifying **-qunroll** without any suboptions is equivalent to **-qunroll=yes**.

-qnounroll is equivalent to **-qunroll=no**.

Usage

The pragma overrides the **-q[no]unroll** compiler option setting for a designated loop. However, even if **#pragma unroll** is specified for a given loop, the compiler remains the final arbiter of whether the loop is actually unrolled.

Only one pragma may be specified on a loop. The pragma must appear immediately before the loop or the **#pragma block_loop** directive to have effect.

The pragma affects only the loop that follows it. An inner nested loop requires a **#pragma unroll** directive to precede it if the desired loop unrolling strategy is different from that of the prevailing **-q[no]unroll** option.

The **#pragma unroll** and **#pragma nounroll** directives can only be used on for loops or **#pragma block_loop** directives. They cannot be applied to do while and while loops.

The loop structure must meet the following conditions:

- There must be only one loop counter variable, one increment point for that variable, and one termination variable. These cannot be altered at any point in the loop nest.
- Loops cannot have multiple entry and exit points. The loop termination must be the only means to exit the loop.
- Dependencies in the loop must not be "backwards-looking". For example, a statement such as `A[i][j] = A[i - 1][j + 1] + 4` must not appear within the loop.

Predefined macros

None.

Examples

In the following example, the **#pragma unroll(3)** directive on the first for loop requires the compiler to replicate the body of the loop three times. The **#pragma unroll** on the second for loop allows the compiler to decide whether to perform unrolling.

```
#pragma unroll(3)
for( i=0; i < n; i++)
{
    a[i] = b[i] * c[i];
}

#pragma unroll
for( j=0; j < n; j++)
{
    a[j] = b[j] * c[j];
}
```

In this example, the first **#pragma unroll(3)** directive results in:

```
i=0;
if (i>n-2) goto remainder;
for (; i<n-2; i+=3) {
    a[i]=b[i] * c[i];
    a[i+1]=b[i+1] * c[i+1];
    a[i+2]=b[i+2] * c[i+2];
}
if (i<n) {
    remainder:
    for (; i<n; i++) {
        a[i]=b[i] * c[i];
    }
}
```

Related information

- “**#pragma block_loop**” on page 274
- “**#pragma loopid**” on page 289

- “`#pragma stream_unroll`” on page 306
- “`#pragma unrollandfuse`” on page 308

-qunwind

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Specifies whether the call stack can be unwound by code looking through the saved registers on the stack.

Specifying **-qnounwind** asserts to the compiler that the stack will not be unwound, and can improve optimization of non-volatile register saves and restores.

Syntax

►► `-q` unwind
nounwind _____ ►►

Defaults

`-qunwind`

Usage

The `setjmp` and `longjmp` families of library functions are safe to use with **-qnounwind**.

► **C++** Specifying **-qnounwind** also implies **-qnoeh**.

Predefined macros

None.

Related information

- “`-qeh` (C++ only) (PPU only)” on page 89

-qupconv (C only)

Category

Portability and migration

Pragma equivalent

`#pragma options [no]upconv`

Purpose

Specifies whether the unsigned specification is preserved when integral promotions are performed.

When **noupconv** is in effect, any unsigned type smaller than an int is converted to int during integral promotions. When **upconv** is in effect, these types are converted to unsigned int during integral promotions.

Syntax

►► — -q noupconv
upconv —►►

Defaults

- **-qnoupconv** for all language levels except **classic** or **extended**
- **-qupconv** when the **classic** or **extended** language levels are in effect

Usage

Sign preservation is provided for compatibility with older dialects of C. The ANSI C standard requires value preservation as opposed to sign preservation.

Predefined macros

None.

Examples

To compile `myprogram.c` so that all unsigned types smaller than int are converted to unsigned int, enter:

invocation `myprogram.c -qupconv`

The following short listing demonstrates the effect of **-qupconv**:

```
#include <stdio.h>
int main(void) {
    unsigned char zero = 0;
    if (-1 < zero)
        printf("Value-preserving rules in effect\n");
    else
        printf("Unsignedness-preserving rules in effect\n");
    return 0;
}
```

Related information

- "Integral and floating-point promotions" in the *XL C/C++ Language Reference*
- `"-qlanglvl"` on page 148

-qutf

Category

Language element control

Pragma equivalent

None.

Purpose

Enables recognition of UTF literal syntax.

Syntax

►► -q noutf
utf _____ ►►

Defaults

- C -qnoutf
- C++ -qutf

Usage

The compiler uses **iconv** to convert the source file to Unicode. If the source file cannot be converted, the compiler will ignore the **-qutf** option and issue a warning.

Predefined macros

None.

Related information

- "UTF literals" in the *XL C/C++ Language Reference*

-v, -V

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.

When the **-v** option is in effect, information is displayed in a comma-separated list. When the **-V** option is in effect, information is displayed in a space-separated list.

Syntax

►► -v
-V _____ ►►

-qversion specified without the **verbose** suboption shows compiler information in the format:

*product_name*Version: *VV.RR.MMMM.LLLL*

where:

V Represents the version.
R Represents the release.
M Represents the modification.
L Represents the level.

Example:

IBM XL C/C++ for Multicore Acceleration for Linux, V10.1
Version: 10.01.0000.0001

-qversion=verbose shows component information in the following format:

component_name Version: *VV.RR(product_name)* Level: *component_level*

where:

component_name
Specifies an installed component, such as the low-level optimizer.
component_level
Represents the level of the installed component.

Example:

IBM XL C/C++ for Multicore Acceleration for Linux, V10.1
Version: 10.01.0000.0001
Driver Version: 10.01(C/C++) Level: 060414
C Front End Version: 10.01(C/C++) Level: 060419
C++ Front End Version: 10.01(C/C++) Level: 060420
High Level Optimizer Version: 10.01(C/C++) and 12.01(Fortran) Level: 060411
Low Level Optimizer Version: 10.01(C/C++) and 12.01(Fortran) Level: 060418

If you want to save this information to the output object file, you can do so with the **-qsaveopt -c** options.

Predefined macros

None.

Related information

- “-qsaveopt” on page 209

-qvrsave (PPU only)

Category

Object code control

Pragma equivalent

#pragma altivec_vrsave

Purpose

Enables code in function prologs and epilogs to maintain the VRSAVE register.

Each bit in the VRSAVE register corresponds to a vector register and, if set to 1, indicates that the corresponding vector register contains data to be saved when a context switch occurs. Use **-qvrsave** to indicate to the compiler that functions in the compilation unit include code needed to maintain the VRSAVE register. Use **-qnovrsave** to indicate to the compiler that functions in the compilation unit do not include code needed to maintain the VRSAVE register.

You can use the pragma to override the current setting of the compiler option for individual functions within your program source.

Syntax

Option syntax

►► — -q vrsave
novrsave —————►►

Pragma syntax

►► — #pragma altivec_vrsave on
off
allon —————►►

Defaults

vrsave: The VRSAVE register is always maintained.

Parameters

on (pragma only)

Function prologs and epilogs include code to maintain the VRSAVE register.

off (pragma only)

Function prologs and epilogs do not include code to maintain the VRSAVE register.

allon (pragma only)

The function containing pragma sets all bits of the VRSAVE register to 1, indicating that all vectors are used and should be saved if a context switch occurs.

Usage

This option and pragma are only supported when **-qaltivec** is in effect.

The pragma can be used only within a function, and its effects apply only to the function in which it appears. Specifying this pragma with different settings within the same function will create an error condition.

Predefined macros

None.

Related information

- “-qaltivec” on page 56

-W

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Suppresses informational, language-level and warning messages.

 This option is equivalent to specifying **-qflag=e : e**.  This option is equivalent to specifying **-qflag=s : s**.

Syntax

►► — -w ————— ◄◄

Defaults

All informational and warning messages are reported.

Usage

Informational and warning messages that supply additional information to a severe error are not disabled by this option.

Predefined macros

None.

Examples

To compile `myprogram.c` so that no warning messages are displayed, enter:

invocation `myprogram.c -w`

The following example shows how informational messages that result from a severe error, in this case caused by problems with overload resolution in C++, are not disabled :

```
void func(int a){}
void func(int a, int b){}
int main(void)
{
    func(1,2,3);
    return 0;
}
```

The output is as follows:

```
"x.cpp", line 6.4: 1540-0218 (S) The call does not match any parameter list for "func".
"x.cpp", line 1.6: 1540-1283 (I) "func(int)" is not a viable candidate.
"x.cpp", line 6.4: 1540-0215 (I) The wrong number of arguments have been specified for "func(int)".
"x.cpp", line 2.6: 1540-1283 (I) "func(int, int)" is not a viable candidate.
"x.cpp", line 6.4: 1540-0215 (I) The wrong number of arguments have been specified for "func(int,
```

Related information

- “-qflag” on page 97
- “-qsuppress” on page 231

-W

Category

Compiler customization

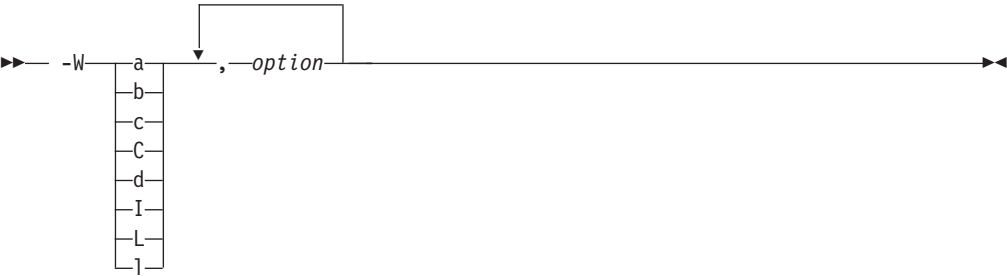
Pragma equivalent

None.

Purpose

Passes the listed options to a component that is executed during compilation.


Syntax



Parameters

option
Any option that is valid for the component to which it is being passed. Spaces must not appear before the *option*.

The following table shows the correspondence between **-W** parameters and the component executable names:

Parameter	Description	Executable name
a	Assembler	ppu-as or spu-as
b	Low-level optimizer	xlCcode
c	Compiler front end	xlcentry, xlCentry
 C	C++ compiler front end	xlCentry
d	Disassembler	dis
I	High-level optimizer, compile step	ipa
L	High-level optimizer, link step	ipa
l	Linker	ppu-ld or spu-ld

Usage

In the string following the **-W** option, use a comma as the separator for each option, and do not include any spaces. If you need to include a character that is special to the shell in the option string, precede the character with a backslash. For example, if you use the **-W** option in the configuration file, you can use the escape sequence backslash comma (\,) to represent a comma in the parameter string.

You do not need the **-W** option to pass most options to the linker **ppu-ld** or **spu-ld**; unrecognized command-line options, except **-q** options, are passed to it automatically. Only linker options with the same letters as compiler options, such as **-v** or **-S**, strictly require **-W**.

Predefined macros

None.

Examples

To compile the file `file.c` and pass the linker option **-berok** to the linker, enter the following command:

```
invocation -Wl,-berok file.c
```

To compile the file `uses_many_symbols.c` and the assembly file `produces_warnings.s` so that `produces_warnings.s` is assembled with the assembler option **-x** (issue warnings and produce cross-reference), and the object files are linked with the option **-s** (write list of object files and strip final executable file), issue the following command:

```
invocation -Wa,-x -Wl,-s produces_warnings.s uses_many_symbols.c
```

Related information

- “Invoking the compiler” on page 1

-qwarn64 (PPU only)

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Enables checking for possible data conversion problems between 32-bit and 64-bit compiler modes.

When **-qwarn64** is in effect, informational messages are displayed where data conversion may cause problems in 64-bit compilation mode, such as:

- Truncation due to explicit or implicit conversion of long types into int types
- Unexpected results due to explicit or implicit conversion of int types into long types
- Invalid memory references due to explicit conversion by cast operations of pointer types into int types

- Invalid memory references due to explicit conversion by cast operations of int types into pointer types
- Problems due to explicit or implicit conversion of constants into long types
- Problems due to explicit or implicit conversion by cast operations of constants into pointer types

Syntax

►► — -q nowarn64
warn64 —————►►

Defaults

-qnowarn64

Usage

This option functions in either 32-bit or 64-bit compiler modes. In 32-bit mode, it functions as a preview aid to discover possible 32-bit to 64-bit migration problems.

Predefined macros

None.

Related information

- -q32, -q64
- “Compiler messages” on page 15

-qxcall

Category

Object code control

Pragma equivalent

None.

Purpose

Generates code to treat static functions within a compilation unit as if they were external functions.

Syntax

►► — -q noxcall
xcall —————►►

Defaults

-qnoxcall

Usage

`-qxcall` generates slower code than `-qnoxcall`.

Predefined macros

None.

Examples

To compile `myprogram.c` so that all static functions are compiled as external functions, enter:

```
invocation myprogram.c -qxcall
```

-qxref

Category

Listings, messages, and compiler information

Pragma equivalent

`#pragma options [no]xref`

Purpose

Produces a compiler listing that includes the cross-reference component of the attribute and cross-reference section of the listing.

When `xref` is in effect, a listing file is generated with a `.lst` suffix for each source file named on the command line. For details of the contents of the listing file, see “Compiler listings” on page 18.

Syntax



Defaults

`-qnoxref`

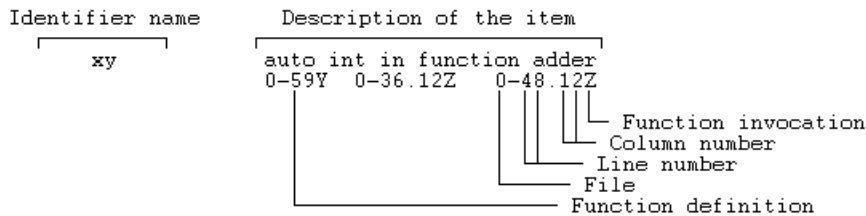
Parameters

full

Reports all identifiers in the program. If you specify `xref` without this suboption, only those identifiers that are used are reported.

Usage

A typical cross-reference listing has the form:



The listing uses the following character codes:

Table 24. Cross-reference listing codes

Character	Meaning
X	Function is declared.
Y	Function is defined.
Z	Function is called.
\$	Type is defined, variable is declared/defined.
#	Variable is assigned to.
&	Variable is defined and initialized.
[blank]	Identifier is referenced.

The **-qnoprint** option overrides this option.

Any function defined with the **#pragma mc_func** directive is listed as being defined on the line of the pragma directive.

Predefined macros

None.

Examples

To compile myprogram.c and produce a cross-reference listing of all identifiers, whether they are used or not, enter:

```
invocation myprogram.c -qxref=full
```

Related information

- “-qattr” on page 62
- “#pragma mc_func” on page 293

-y

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.

Syntax



Defaults

- `-yn`
- `-yz` for compilation targeting the SPU, for single-precision arithmetic only

Parameters

m Round toward minus infinity.

Note: This suboption is not supported on the SPU for single-precision arithmetic.

n Round to the nearest representable number, ties to even.

Note: This suboption is not supported on the SPU for single-precision arithmetic.

p Round toward plus infinity.

Note: This suboption is not supported on the SPU for single-precision arithmetic.

z Round toward zero.

Usage

If your program contains operations involving long doubles, the rounding mode must be set to `-yn` (round-to-nearest representable number, ties to even).

For single-precision arithmetic operations, the SPU supports only round-to-zero mode. Therefore, compile-time folding will be done in round-to-zero, regardless of the setting of the `-y` option. However, for compile-time single/double precision conversions, as well as evaluation of single- (and double-) precision literals, the SPU does respond to the rounding mode specified by the `-y` option.

Predefined macros

None.

Examples

To compile `myprogram.c` so that constant floating-point expressions are rounded toward zero at compile time, enter:

invocation `myprogram.c -yz`

-Z

Category

Linking

Pragma equivalent

None.

Purpose

Specifies a prefix for the library search path to be used by the linker.

Syntax

►► — *-Z—string—* ————— ►►

Defaults

By default, the linker searches the /usr/lib/ directory for library files.

Parameters

string

Represents the prefix to be added to the directory search path for library files.

Predefined macros

None.

Chapter 4. Compiler pragmas reference

The following sections describe the pragmas available:

- “Pragma directive syntax”
- “Scope of pragma directives” on page 270
- “Summary of compiler pragmas by functional category” on page 270
- “Individual pragma descriptions” on page 274

Pragma directive syntax

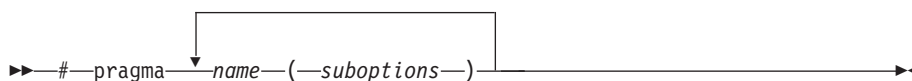
XL C/C++ supports three forms of pragma directives:

#pragma options *option_name*

These pragmas use exactly the same syntax as their command-line option equivalent. The exact syntax and list of supported pragmas of this type are provided in “#pragma options” on page 294.

#pragma *name*

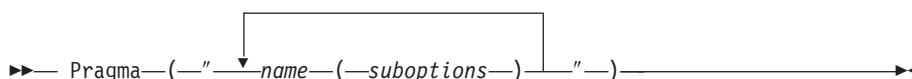
This form uses the following syntax:



The *name* is the pragma directive name, and the *suboptions* are any required or optional suboptions that can be specified for the pragma, where applicable.

_Pragma ("name")

This form uses the following syntax:



For example, the statement:

```
_Pragma ( "pack(1)" )
```

is equivalent to:

```
#pragma pack(1)
```

For all forms of pragma statements, you can specify more than one *name* and *suboptions* in a single **#pragma** statement.

The *name* on a pragma is subject to macro substitutions, unless otherwise stated. The compiler ignores unrecognized pragmas, issuing an informational message indicating this.

If you have any pragmas that are not common to both C and C++ in code that will be compiled by both compilers, you may add conditional compilation directives around the pragmas. (This is not strictly necessary since unrecognized pragmas are

ignored.) For example, **#pragma object_model** is only recognized by the C++ compiler, so you may decide to add conditional compilation directives around the pragma.

```
#ifdef __cplusplus
#pragma object_model(pop)
#endif
```

Scope of pragma directives

Many pragma directives can be specified at any point within the source code in a compilation unit; others must be specified before any other directives or source code statements. In the individual descriptions for each pragma, the "Usage" section describes any constraints on the pragma's placement.

In general, if you specify a pragma directive before any code in your source program, it applies to the entire compilation unit, including any header files that are included. For a directive that can appear anywhere in your source code, it applies from the point at which it is specified, until the end of the compilation unit.

You can further restrict the scope of a pragma's application by using complementary pairs of pragma directives around a selected section of code. For example, using **#pragma options source** and **#pragma options nosource** directives as follows requests that only the selected parts of your source code be included in your compiler listing:

```
#pragma options source

/* Source code between the source and nosource pragma
   options is included in the compiler listing          */

#pragma options nosource
```

Many pragmas provide "pop" or "reset" suboptions that allow you to enable and disable pragma settings in a stack-based fashion; examples of these are provided in the relevant pragma descriptions.

Summary of compiler pragmas by functional category

The XL C/C++ pragmas available are grouped into the following categories:

- "Language element control" on page 271
- "C++ template pragmas" on page 271
- "Floating-point and integer control" on page 271
- "Error checking and debugging" on page 271
- "Listings, messages and compiler information" on page 272
- "Optimization and tuning" on page 272
- "Object code control" on page 273
- "Portability and migration" on page 273
- "Compiler customization" on page 274

For descriptions of these categories, see "Summary of compiler options by functional category" on page 31.

Language element control

Table 25. Language element control pragmas

Pragma	Description
#pragma langlvl (C only)	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.
"#pragma mc_func" on page 293	Allows you to embed a short sequence of machine instructions "inline" within your program source code.
"#pragma options" on page 294	Specifies compiler options in your source program.

C++ template pragmas

Table 26. C++ template pragmas

Pragma	Description
"#pragma define, #pragma instantiate (C++ only)" on page 279	Provides an alternative method for explicitly instantiating a template class.
"#pragma do_not_instantiate (C++ only)" on page 281	Prevents the specified template declaration from being instantiated.
"#pragma implementation (C++ only)" on page 287	For use with the -qtempinc compiler option, supplies the name of the file containing the template definitions corresponding to the template declarations contained in a header file.

Floating-point and integer control

Table 27. Floating-point and integer control pragmas

Pragma	Description
#pragma chars	Determines whether all variables of type char are treated as either signed or unsigned.
#pragma enum	Specifies the amount of storage occupied by enumerations.

Error checking and debugging

Table 28. Error checking and debugging pragmas

Pragma	Description
"#pragma ibm snapshot" on page 286	Specifies a location at which a breakpoint can be set and defines a list of variables that can be examined when program execution reaches that location.
#pragma info	Produces or suppresses groups of informational messages.

Listings, messages and compiler information

Table 29. Listings, messages and compiler information pragmas

Pragma	Description
"#pragma report (C++ only)" on page 304	Controls the generation of diagnostic messages.

Optimization and tuning

Table 30. Optimization and tuning pragmas

Pragma	Description
"#pragma block_loop" on page 274	Marks a block with a scope-unique identifier.
"#pragma STDC cx_limited_range" on page 305	Instructs the compiler that complex division and absolute value are only invoked with values such that intermediate calculation will not overflow or lose significance.
"#pragma disjoint" on page 280	Lists identifiers that are not aliased to each other within the scope of their use.
"#pragma execution_frequency" on page 282	Marks program source code that you expect will be either very frequently or very infrequently executed.
"#pragma expected_value" on page 284	Specifies the value that a parameter passed in a function call is most likely to take at run time. The compiler can use this information to perform certain optimizations, such as function cloning and inlining.
#pragma isolated_call	Specifies functions in the source file that have no side effects other than those implied by their parameters.
"#pragma leaves" on page 288	Informs the compiler that a named function never returns to the instruction following a call to that function.
"#pragma loopid" on page 289	Marks a block with a scope-unique identifier.
#pragma nosimd	When used with -qhot=simd , disables the generation of SIMD instructions for the next loop.
#pragma novector	When used with -qhot=novector , disables auto-vectorization of the next loop.
"#pragma option_override" on page 296	Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.
"#pragma reachable" on page 302	Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.
"#pragma reg_killed_by" on page 303	Specifies registers that may be altered by functions specified by #pragma mc_func .
"#pragma stream_unroll" on page 306	When optimization is enabled, breaks a stream contained in a for loop into multiple streams.

Table 30. Optimization and tuning pragmas (continued)

Pragma	Description
#pragma unroll	Controls loop unrolling, for improved performance.
"#pragma unrollandfuse" on page 308	Instructs the compiler to attempt an unroll and fuse operation on nested for loops.

Object code control

Table 31. Object code control pragmas

Pragma	Description
#pragma alloca (C only)	Provides an inline definition of system function alloca when it is called from source code that does not include the alloca.h header.
"#pragma comment" on page 278	Places a comment into the object module.
"#pragma hashome (C++ only)" on page 285	Informs the compiler that the specified class has a home module that will be specified by #pragma ishome .
"#pragma ishome (C++ only)" on page 288	Informs the compiler that the specified class's home module is the current compilation unit.
"#pragma map" on page 290	Converts all references to an identifier to another, externally defined identifier.
"#pragma pack" on page 298	Sets the alignment of all aggregate members to a specified byte boundary.
#pragma priority (C++ only)	Specifies the priority level for the initialization of static objects.
"#pragma reg_killed_by" on page 303	Specifies registers that may be altered by functions specified by #pragma mc_func .
#pragma strings	Specifies the storage type for string literals.
"#pragma weak" on page 309	Prevents the linker from issuing error messages if it encounters a symbol multiply-defined during linking, or if it does not find a definition for a symbol.

Portability and migration

Table 32. Portability and migration pragmas

Pragma	Description
#pragma align	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.

Compiler customization

Table 33. Compiler customization pragmas

Pragma	Description
<code>#pragma complexgcc</code>	Specifies whether to use GCC parameter-passing conventions for complex data types (equivalent to enabling <code>-qfloat=complexgcc</code>).

Individual pragma descriptions

This section contains descriptions of individual pragmas available in XL C/C++.

For each pragma, the following information is given:

Category

The functional category to which the pragma belongs is listed here.

Purpose

This section provides a brief description of the effect of the pragma, and why you might want to use it.

Syntax

This section provides the syntax for the pragma. For convenience, the **#pragma name** form of the directive is used in each case. However, it is perfectly valid to use the alternate C99-style `_Pragma` operator syntax; see “Pragma directive syntax” on page 269 for details.

Parameters

This section describes the suboptions that are available for the pragma, where applicable.

Usage This section describes any rules or usage considerations you should be aware of when using the pragma. These can include restrictions on the pragma’s applicability, valid placement of the pragma, and so on.

Examples

Where appropriate, examples of pragma directive use are provided in this section.

#pragma align

See “-qalign” on page 53.

#pragma alloca (C only)

See “-qalloca, -ma (C only)” on page 55.

#pragma block_loop

Category

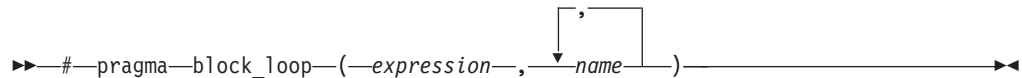
Optimization and tuning

Purpose

Marks a block with a scope-unique identifier.

Syntax

►► `#pragma block_loop (expression , name)` ►►



Parameters

expression

An integer expression representing the size of the iteration group.

name

An identifier that is unique within the scoping unit. If you do not specify a *name*, blocking occurs on the first for loop or loop following the **#pragma block_loop** directive.

Usage

For loop blocking to occur, a **#pragma block_loop** directive must precede a for loop.

If you specify **#pragma unroll**, **#pragma unrollandfuse** or **#pragma stream_unroll** for a blocking loop, the blocking loop is unrolled, unrolled and fused or stream unrolled respectively, if the blocking loop is actually created. Otherwise, this directive has no effect.

If you specify **#pragma unrollandfuse**, **#pragma unroll** or **#pragma stream_unroll** directive for a blocked loop, the directive is applied to the blocked loop after the blocking loop is created. If the blocking loop is not created, this directive is applied to the loop intended for blocking, as if the corresponding **#pragma block_loop** directive was not specified.

You must not specify **#pragma block_loop** more than once, or combine the directive with **#pragma nounroll**, **#pragma unroll**, **#pragma nounrollandfuse**, **#pragma unrollandfuse**, or **#pragma stream_unroll** directives for the same for loop. Also, you should not apply more than one **#pragma unroll** directive to a single block loop directive.

Processing of all **#pragma block_loop** directives is always completed before performing any unrolling indicated by any of the unroll directives

Examples

The following two examples show the use of **#pragma block_loop** and **#pragma loop_id** for loop tiling:

```
#pragma block_loop(50, mymainloop)
#pragma block_loop(20, myfirstloop, mysecondloop)
#pragma loop_id(mymainloop)
    for (i=0; i < n; i++)
    {
        #pragma loop_id(myfirstloop)
        for (j=0; j < m; j++)
        {
            #pragma loop_id(mysecondloop)
            for (k=0; k < m; k++)
            {
```

```

        ...
    }
}

#pragma block_loop(50, mymainloop)
#pragma block_loop(20, myfirstloop, mysecondloop)
#pragma loopid(mymainloop)
    for (i=0; i < n; i++)
    {
        #pragma loopid(myfirstloop)
            for (j=0; j < m; j++)
            {
                #pragma loopid(mysecondloop)
                    for (k=0; k < m; k++)
                    {
                        ...
                    }
            }
    }
}

```

The following example shows the use **#pragma block_loop** and **#pragma loop_id** for loop interchange.

```

    for (i=0; i < n; i++)
    {
        for (j=0; j < n; j++)
        {
            #pragma block_loop(1,myloop1)
                for (k=0; k < m; k++)
                {
                    #pragma loopid(myloop1)
                        for (l=0; l < m; l++)
                        {
                            ...
                        }
                }
        }
    }
}

```

The following example shows the use of **#pragma block_loop** and **#pragma loop_id** for loop tiling for multi-level memory hierarchy:

```

#pragma block_loop(l3factor, first_level_blocking)
    for (i=0; i < n; i++)
    {
        #pragma loopid(first_level_blocking)
        #pragma block_loop(l2factor, inner_space)
            for (j=0; j < n; j++)
            {
                #pragma loopid(inner_space)
                    for (k=0; k < m; k++)
                    {
                        for (l=0; l < m; l++)
                        {
                            ...
                        }
                    }
            }
    }
}

```

The following example uses **#pragma unrollandfuse** and **#pragma block_loop** to unroll and fuse a blocking loop.

```

#pragma unrollandfuse
#pragma block_loop(10)
    for (i = 0; i < N; ++i) {
    }
}

```

In this case, if the block loop directive is ignored, the unroll directives have no effect.

The following example shows the use of **#pragma unroll** and **#pragma block_loop** to unroll a blocked loop.

```
#pragma block_loop(10)
#pragma unroll(2)
for (i = 0; i < N; ++i) {
}
```

In this case, if the block loop directive is ignored, the unblocked loop is still subjected to unrolling. If blocking does happen, the unroll directive is applied to the blocked loop.

The following examples show invalid uses of the directive. The first example shows **#pragma block_loop** used on an undefined loop identifier:

```
#pragma block_loop(50, myloop)
for (i=0; i < n; i++)
{
}
```

Referencing myloop is not allowed, since it is not in the nest and may not be defined.

In the following example, referencing myloop is not allowed, since it is not in the same loop nest:

```
for (i=0; i < n; i++)
{
  #pragma loopid(myLoop)
  for (j=0; j < i; j++)
  {
    ...
  }
}
#pragma block_loop(myLoop)
for (i=0; i < n; i++)
{
  ...
}
```

The following examples are invalid since the unroll directives conflict with each other:

```
#pragma unrollandfuse(5)
#pragma unroll(2)
#pragma block_loop(10)
for (i = 0; i < N; ++i) {
}

#pragma block_loop(10)
#pragma unroll(5)
#pragma unroll(10)
for (i = 0; i < N; ++i) {
}
```

Related information

- “#pragma loopid” on page 289
- “-qunroll” on page 252
- “#pragma unrollandfuse” on page 308
- “#pragma stream_unroll” on page 306

#pragma chars

See “-qchars” on page 69.

#pragma comment

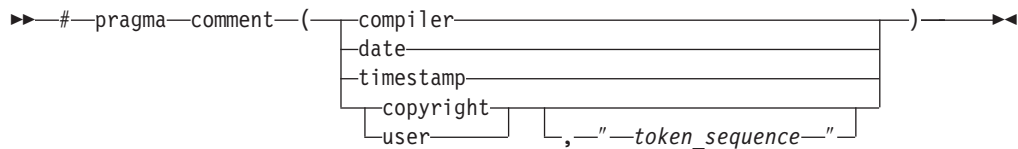
Category

Object code control

Purpose

Places a comment into the object module.

Syntax



Parameters

compiler

Appends the name and version of the compiler at the end of the generated object module.

date

The date and time of the compilation are appended at the end of the generated object module.

timestamp

Appends the date and time of the last modification of the source at the end of the generated object module.

copyright

Places the text specified by the *token_sequence*, if any, into the generated object module. The *token_sequence* is included in the generated executable and loaded into memory when the program is run.

user

Places the text specified by the *token_sequence*, if any, into the generated object module. The *token_sequence* is included in the generated executable but is *not* loaded into memory when the program is run.

token_sequence

The characters in this field, if specified, must be enclosed in double quotation marks (`"`). If the string literal specified in the *token_sequence* exceeds 32 767 bytes, an information message is emitted and the pragma is ignored.

Usage

More than one **comment** directive can appear in a translation unit, and each type of **comment** directive can appear more than once, with the exception of **copyright**, which can appear only once.

You can display the object-file comments by using the operating system **strings** command.

Examples

Assume that following program code is compiled to produce output file a.out:

```
#pragma comment(date)
#pragma comment(compiler)
#pragma comment(timestamp)
#pragma comment(copyright,"My copyright")

int main() {

return 0;
}
```

Issuing the command:

```
strings a.out
```

will cause the comment information embedded in a.out to be displayed, along with any other strings that may be found in a.out. For example, assuming the program code shown above:

```
Mon Mar 1 10:28:03 2008
XL C/C++ for Multicore Acceleration Compiler Version 10.1
Mon Mar 1 10:28:09 2008
My copyright
```

#pragma complexgcc

See “-qcomplexgccincl” on page 74.

#pragma define, #pragma instantiate (C++ only)

Category

Template control

Purpose

Provides an alternative method for explicitly instantiating a template class.

Syntax

```
➤ #pragma define  
instantiate (—template_class_name—) ➤
```

Parameters

template_class_name

The name of the template class to be instantiated.

Usage

This pragma provides equivalent functionality to standard C++ explicit instantiation, and is provided for backwards compatibility purposes only. New applications should use standard C++ explicit instantiation.

The pragma can appear anywhere an explicit instantiation statement can appear.

Examples

The following directive:

```
#pragma define(Array<char>)
```

is equivalent to the following explicit instantiation:

```
template class Array<char>;
```

Related information

- "Explicit instantiation" in the *XL C/C++ Language Reference*
- "#pragma do_not_instantiate (C++ only)" on page 281

#pragma disjoint

Category

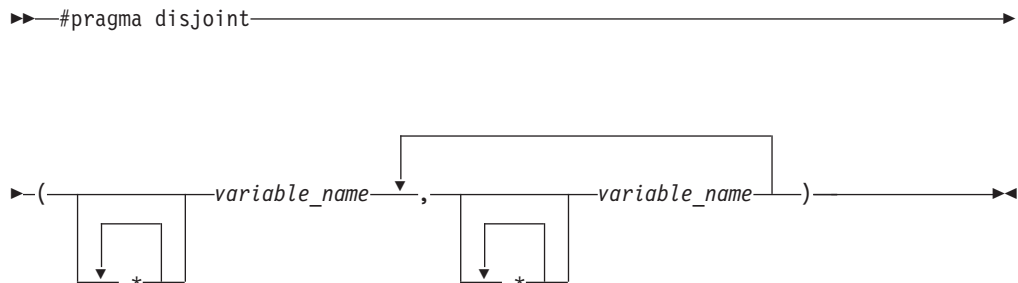
Optimization and tuning

Purpose

Lists identifiers that are not aliased to each other within the scope of their use.

By informing the compiler that none of the identifiers listed in the pragma shares the same physical storage, the pragma provides more opportunity for optimizations.

Syntax



Parameters

variable_name

The name of a variable. It must not refer to any of the following:

- A member of a structure, class, or union
- A structure, union, or enumeration tag
- An enumeration constant
- A typedef name
- A label

Usage

The **#pragma disjoint** directive asserts that none of the identifiers listed in the pragma share physical storage; if any the identifiers *do* actually share physical storage, the pragma may give incorrect results.

The pragma can appear anywhere in the source program that a declaration is allowed. An identifier in the directive must be visible at the point in the program where the pragma appears.

You must declare the identifiers before using them in the pragma. Your program must not dereference a pointer in the identifier list nor use it as a function argument before it appears in the directive.

This pragma can be disabled with the **-qignprag** compiler option.

Examples

The following example shows the use of **#pragma disjoint**.

```
int a, b, *ptr_a, *ptr_b;

#pragma disjoint(*ptr_a, b)  /* *ptr_a never points to b */
#pragma disjoint(*ptr_b, a)  /* *ptr_b never points to a */
one_function()
{
    b = 6;
    *ptr_a = 7;  /* Assignment will not change the value of b */

    another_function(b);  /* Argument "b" has the value 6 */
}
```

External pointer `ptr_a` does not share storage with and never points to the external variable `b`. Consequently, assigning 7 to the object to which `ptr_a` points will not change the value of `b`. Likewise, external pointer `ptr_b` does not share storage with and never points to the external variable `a`. The compiler can assume that the argument to `another_function` has the value 6 and will not reload the variable from memory.

#pragma do_not_instantiate (C++ only)

Category

Template control

Purpose

Prevents the specified template declaration from being instantiated.

You can use this pragma to suppress the implicit instantiation of a template for which a definition is supplied.

Syntax

►►—#—pragma—do_not_instantiate—*template_class_name*—————►►

Parameters

template_class_name

The name of the template class that should not be instantiated.

Usage

If you are handling template instantiations manually (that is, **-qnotempinc** and **-qnotemplateregistry** are specified), and the specified template instantiation already exists in another compilation unit, using **#pragma do_not_instantiate** ensures that you do not get multiple symbol definitions during the link step.

You can also use the **-qtmplinst** option to suppress implicit instantiation of template declarations for multiple compilation units. See “-qtmplinst (C++ only)” on page 247.

Examples

The following shows the usage of the pragma:

```
#pragma do_not_instantiate Stack < int >
```

Related information

- “#pragma define, #pragma instantiate (C++ only)” on page 279
- “-qtmplinst (C++ only)” on page 247
- “Explicit instantiation” in the *XL C/C++ Language Reference*
- “-qtempinc (C++ only)” on page 239
- “-qtemplateregistry (C++ only)” on page 242

#pragma enum

See “-qenum” on page 90.

#pragma execution_frequency

Category

Optimization and tuning

Purpose

Marks program source code that you expect will be either very frequently or very infrequently executed.

When optimization is enabled, the pragma is used as a hint to the optimizer.

Syntax

►► #pragma execution_frequency (very_low very_high) ►►

Parameters

very_low

Marks source code that you expect will be executed very infrequently.

very_high

Marks source code that you expect will be executed very frequently.

Usage

Use this pragma in conjunction with an optimization option; if optimization is not enabled, the pragma has no effect.

The pragma must be placed within block scope, and acts on the closest point of branching.

Examples

In the following example, the pragma is used in an if statement block to mark code that is executed infrequently.

```
int *array = (int *) malloc(10000);

if (array == NULL) {
    /* Block A */
    #pragma execution_frequency(very_low)
    error();
}
```

In the next example, the code block Block B is marked as infrequently executed and Block C is likely to be chosen during branching.

```
if (Foo > 0) {
    #pragma execution_frequency(very_low)
    /* Block B */
    doSomething();
} else {
    /* Block C */
    doAnotherThing();
}
```

In this example, the pragma is used in a switch statement block to mark code that is executed frequently.

```
while (counter > 0) {
    #pragma execution_frequency(very_high)
    doSomething();
} /* This loop is very likely to be executed. */

switch (a) {
    case 1:
        doOneThing();
        break;
    case 2:
        #pragma execution_frequency(very_high)
        doTwoThings();
        break;
    default:
        doNothing();
} /* The second case is frequently chosen. */
```

The following example shows how the pragma must be applied at block scope and affects the closest branching.

```
int a;
#pragma execution_frequency(very_low)
int b;

int foo(boolean boo) {
    #pragma execution_frequency(very_low)
    char c;

    if (boo) {
        /* Block A */
        doSomething();
        {
            /* Block C */
            doSomethingAgain();
            #pragma execution_frequency(very_low)
        }
    }
}
```

```

        doAnotherThing();
    }
} else {
    /* Block B */
    doNothing();
}

return 0;
}

#pragma execution_frequency(very_low)

```

#pragma expected_value

Category

Optimization and tuning

Purpose

Specifies the value that a parameter passed in a function call is most likely to take at run time. The compiler can use this information to perform certain optimizations, such as function cloning and inlining.

Syntax

```

▶▶—#pragma expected_value—(—argument—,—value—)—————▶▶

```

Parameters

argument

The name of the parameter for which you want to provide the expected value. The parameter must be of a simple built-in integral, Boolean, character, or floating-point type.

value

A constant literal representing the value that you expect will most likely be taken by the parameter at run time. *value* can be an expression as long as it is a compile time constant expression.

Usage

The directive must appear inside the body of a function definition, before the first statement (including declaration statements). It is not supported within nested functions.

If you specify an expected value of a type different from that of the declared type of the parameter variable, the value will be implicitly converted only if allowed. Otherwise, a warning is issued.

For each parameter that will be provided the expected value there is a limit of one directive. Parameters that will not be provided the expected value do not require a directive.

Examples

The following example tells the compiler that the most likely values for parameters *a* and *b* are 1 and 0, respectively:

```
int func(int a,int b)
{
#pragma expected_value(a,1)
#pragma expected_value(b,0)
...
...
}
```

Related information

- “`#pragma execution_frequency`” on page 282

#pragma hashome (C++ only)

Category

Object code control

Purpose

Informs the compiler that the specified class has a home module that will be specified by **#pragma ishome**.

This class’s virtual function table, along with certain inline functions, will not be generated as static. Instead, they will be referenced as externals in the compilation unit of the class in which **#pragma ishome** is specified.

Syntax

```
▶▶ #pragma hashome (—class_name—  
└allinlines┘) ─────────────────▶▶
```

Parameters

class_name

The name of a class to be referenced externally. *class_name* must be a class and it must be defined.

allinlines

Specifies that all inline functions from within *class_name* should be referenced as being external.

Usage

A warning will be produced if there is a **#pragma ishome** without a matching **#pragma hashome**.

Examples

In the following example, compiling the code samples will generate virtual function tables and the definition of `S::foo()` only for compilation unit `a.o`, but not for `b.o`. This reduces the amount of code generated for the application.

```
// a.h
struct S
{
    virtual void foo() {}

    virtual void bar();
};
```

```
// a.C
#pragma ishome(S)
#pragma hashome (S)

#include "a.h"

int main()
{
    S s;
    s.foo();
    s.bar();
}
```

```
// b.C
#pragma hashome(S)
#include "a.h"

void S::bar() {}
```

Related information

- “`#pragma ishome` (C++ only)” on page 288

#pragma ibm snapshot

Category

Error checking and debugging

Purpose

Specifies a location at which a breakpoint can be set and defines a list of variables that can be examined when program execution reaches that location.

You can use this pragma to facilitate debugging optimized code produced by the compiler.

Syntax

```
▶▶ #pragma ibm snapshot ( variable_name ) ▶▶
```

Parameters

variable_name

A variable name. It must not refer to structure, class, or union members.

Usage

During a debugging session, you can place a breakpoint on the line at which the directive appears, to view the values of the named variables. When you compile with optimization and the **-g** option, the named variables are guaranteed to be visible to the debugger.

This pragma does not consistently preserve the contents of variables with a static storage class at high optimization levels. Variables specified in the directive should be considered read-only while being observed in the debugger, and should not be modified. Modifying these variables in the debugger may result in unpredictable behavior.

Examples

```
#pragma ibm snapshot(a, b, c)
```

Related information

- “-g” on page 108
- “-O, -qoptimize” on page 179

#pragma implementation (C++ only)

Category

Template control

Purpose

For use with the **-qtempinc** compiler option, supplies the name of the file containing the template definitions corresponding to the template declarations contained in a header file.

Syntax

►► #pragma implementation (—"file_name"—) ◀◀

Parameters

file_name

The name of the file containing the definitions for members of template classes declared in the header file.

Usage

This pragma is not normally required if your template implementation file has the same name as the header file containing the template declarations, and a .c extension. You only need to use the pragma if the template implementation file does not conform to this file-naming convention. For more information about using template implementation files, see "Using C++ templates"

#pragma implementation is only effective if the **-qtempinc** option is in effect. Otherwise, the pragma has no meaning and is ignored.

The pragma can appear in the header file containing the template declarations, or in a source file that includes the header file. It can appear anywhere that a declaration is allowed.

Related information

- “-qtempinc (C++ only)” on page 239
- "Using C++ templates"

#pragma info

See “-qinfo” on page 124.

#pragma ishome (C++ only)

Category

Object code control

Purpose

Informs the compiler that the specified class’s home module is the current compilation unit.

The home module is where items, such as the virtual function table, are stored. If an item is referenced from outside of the compilation unit, it will not be generated outside its home. This can reduce the amount of code generated for the application.

Syntax

►►—#—pragma—ishome—(*—class_name—*)—►►

Parameters

class_name

The name of the class whose home will be the current compilation unit.

Usage

A warning will be produced if there is a **#pragma ishome** without a matching **#pragma hashome**.

Examples

See “#pragma hashome (C++ only)” on page 285

Related information

- “#pragma hashome (C++ only)” on page 285

#pragma isolated_call

See “-qisolated_call” on page 141.

#pragma langlvl (C only)

See “-qlanglvl” on page 148.

#pragma leaves

Category

Optimization and tuning

Purpose

Informs the compiler that a named function never returns to the instruction following a call to that function.

By informing the compiler that it can ignore any code after the function, the directive allows for additional opportunities for optimization.

This pragma is commonly used for custom error-handling functions, in which programs can be terminated if a certain error is encountered.

Note: The compiler automatically inserts **#pragma leaves** directives for calls to the `longjmp` family of functions (`longjmp`, `_longjmp`, `siglongjmp`, and `_siglongjmp`) when you include the `setjmp.h` header.

Syntax



```
➡ #pragma leaves (function_name,) ➡
```

Parameters

function_name

The name of the function that does not return to the instruction following the call to it.

Defaults

Not applicable.

Examples

```
#pragma leaves(handle_error_and_quit)
void test_value(int value)
{
    if (value == ERROR_VALUE)
    {
        handle_error_and_quit(value);
        TryAgain(); // optimizer ignores this because
                   // never returns to execute it
    }
}
```

Related information

- “#pragma reachable” on page 302.

#pragma loopid

Category

Optimization and tuning

Purpose

Marks a block with a scope-unique identifier.

Syntax

►► `#pragma loopid (—name—)` ◄◄

Parameters

name

An identifier that is unique within the scoping unit.

Usage

The **#pragma loopid** directive must immediately precede a **#pragma block_loop** directive or for loop. The specified name can be used by **#pragma block_loop** to control transformations on that loop. It can also be used to provide information on loop transformations through the use of the **-qreport** compiler option.

You must not specify **#pragma loopid** more than once for a given loop.

Examples

For examples of **#pragma loopid** usage, see “**#pragma block_loop**” on page 274.

Related information

- “**-qunroll**” on page 252
- “**#pragma block_loop**” on page 274
- “**#pragma unrollandfuse**” on page 308

#pragma map

Category

Object code control

Purpose

Converts all references to an identifier to another, externally defined identifier.

Syntax

#pragma map syntax – C



►► `#pragma map (—name1—, —"name2"—)` ◄◄

#pragma map syntax – C++

►► `#pragma map (—name1—(—argument_list—), —"name2"—)` ◄◄

Parameters

name1

The name used in the source code.  *name1* can represent a data object or function with external linkage.  *name1* can represent a data object, a

non-overloaded or overloaded function, or overloaded operator, with external linkage. If the name to be mapped is not in the global namespace, it must be fully qualified.

name1 should be declared in the same compilation unit in which it is referenced, but should not be defined in any other compilation unit. *name1* must not be used in another **#pragma map** directive or any assembly label declaration anywhere in the program.

► C++ *argument_list*

The list of arguments for the overloaded function or operator function designated by *name1*. If *name1* designates an overloaded function, the function must be parenthesized and must include its argument list if it exists. If *name1* designates a non-overloaded function, only *name1* is required, and the parentheses and argument list are optional.

name2

The name that will appear in the object code. ► C *name2* can represent a data object or function with external linkage.

► C++ *name2* can represent a data object, a non-overloaded or overloaded function, or overloaded operator, with external linkage. *name2* must be specified using its mangled name. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file. (See also "Name mangling" in the *XL C/C++ Language Reference* for details on using the extern "C" linkage specifier on declarations to prevent name mangling.)

If the name exceeds 65535 bytes, an informational message is emitted and the pragma is ignored.

name2 may or may not be declared in the same compilation unit in which *name1* is referenced, but must not be defined in the same compilation unit. Also, *name2* should not be referenced anywhere in the compilation unit where *name1* is referenced. *name2* must not be the same as that used in another **#pragma map** directive or any assembly label declaration in the same compilation unit.

Usage

The **#pragma map** directive can appear anywhere in the program. Note that in order for a function to be actually mapped, the map target function (*name2*) must have a definition available at link time (from another compilation unit), and the map source function (*name1*) must be called in your program.

You cannot use **#pragma map** with compiler built-in functions.

Examples

The following is an example of **#pragma map** used to map a function name (using the mangled name for the map name in C++):

```
/* Compilation unit 1: */

#include <stdio.h>

void foo();
extern void bar(); /* optional */

#if __cplusplus
```

```

#pragma map (foo, "_Z3barv")
#else#pragma map (foo, "bar")
#endif
int main()
{
    foo();
}

/* Compilation unit 2: */


#include <stdio.h>

void bar()
{
    printf("Hello from foo bar!\n");
}

```

The call to `foo` in compilation unit 1 resolves to a call to `bar`:

Hello from foo bar!

 The following is an example of **#pragma map** used to map an overloaded function name (using C linkage, to avoid using the mangled name for the map name):

```

// Compilation unit 1:

#include <iostream>
#include <string>

using namespace std;

void foo();
void foo(const string&);
extern "C" void bar(const string&); // optional

#pragma map (foo(const string&), "bar")

int main()
{
    foo("Have a nice day!");
}

// Compilation unit 2:

#include <iostream>
#include <string>

using namespace std;

extern "C" void bar(const string& s)
{
    cout << "Hello from foo bar!" << endl;
    cout << s << endl;
}

```

The call to `foo(const string&)` in compilation unit 1 resolves to a call to `bar(const string&)`:

Hello from foo bar!
Have a nice day!

Related information

- "Assembly labels" in the *XL C/C++ Language Reference*

#pragma mc_func

Category

Language element control

Purpose

Allows you to embed a short sequence of machine instructions "inline" within your program source code.

The pragma instructs the compiler to generate specified instructions in place rather than the usual linkage code. Using this pragma avoids performance penalties associated with making a call to an assembler-coded external function. This pragma is similar in function to inline asm statements supported in this and other compilers; see "Inline assembly statements" in the *XL C/C++ Language Reference* for more information.

Syntax

```
▶▶ #pragma mc_func function_name { instruction_sequence } ▶▶
```

Parameters

function_name

The name of a previously-defined function containing machine instructions. If the function is not previously-defined, the compiler will treat the pragma as a function definition.

instruction_sequence

A string containing a sequence of zero or more hexadecimal digits. The number of digits must comprise an integral multiple of 32 bits. If the string exceeds 16384 bytes, a warning message is emitted and the pragma is ignored.

Usage

This pragma defines a function and should appear in your program source only where functions are ordinarily defined.

The compiler passes parameters to the function in the same way as to any other function. For example, in functions taking integer-type arguments, the first parameter is passed to GPR3, the second to GPR4, and so on. Values returned by the function will be in GPR3 for integer values, and FPR1 for float or double values.

Code generated from *instruction_sequence* may use any and all volatile registers available on your system unless you use **#pragma reg_killed_by** to list a specific register set for use by the function. See "**#pragma reg_killed_by**" on page 303 for a list of volatile registers available on your system.

Inlining options do not affect functions defined by **#pragma mc_func**. However, you may be able to improve runtime performance of such functions with **#pragma isolated_call**.

Note: This pragma should only be used in sections of code that are not inside parallel regions.

Examples

In the following example, **#pragma mc_func** is used to define a function called `add_logical`. The function consists of machine instructions to add 2 integers with so-called *end-around carry*; that is, if a carry out results from the add then add the carry to the sum. This formula is frequently used in checksum computations.

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
/*      addc      r3 <- r3, r4      */
/*      addze     r3 <- r3, carry bit */

main() {

    int i,j,k;

    i = 4;
    k = -4;
    j = add_logical(i,k);
    printf("\n\nresult = %d\n\n",j);
}
```

The result of running the program is:

```
result = 1
```

Related information

- “-qisolated_call” on page 141
- “#pragma reg_killed_by” on page 303
- “Inline assembly statements” in the *XL C/C++ Language Reference*

#pragma nosimd

See “-qhot” on page 115.

#pragma novector

See “-qhot” on page 115.

#pragma options

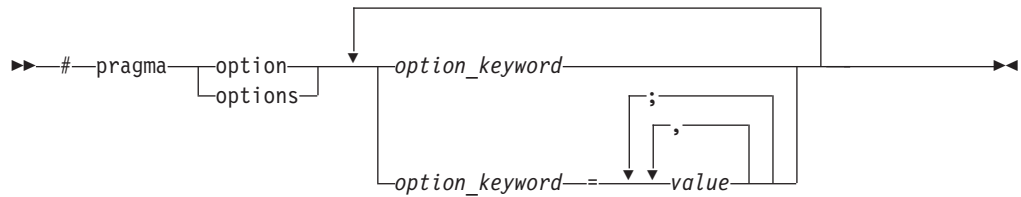
Category

Language element control

Purpose


Specifies compiler options in your source program.




Syntax



Parameters

The settings in the table below are valid *options* for **#pragma options**. For more information, refer to the pages of the equivalent compiler option.

Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent
align= <i>option</i>	"-qalign" on page 53
[no]attr	"-qattr" on page 62
attr=full	
chars= <i>option</i>	"-qchars" on page 69
[no]check	"-qcheck" on page 70
[no]compact	"-qcompact" on page 74
[no]dbcs	"-qmbcs, -qdbcs" on page 173
[no]digraph	"-qdigraph" on page 83
[no]dollar	"-qdollar" on page 85
enum= <i>option</i>	"-qenum" on page 90
flag= <i>option</i>	"-qflag" on page 97
float=[no] <i>option</i>	"-qfloat" on page 98
[no]flttrap= <i>option</i>	"-qflttrap (PPU only)" on page 103
[no]fullpath	"-qfullpath" on page 106
halt	"-qhalt" on page 113
[no]idirfirst	"-qidirfirst" on page 119
[no]ignerrno	"-qignerrno" on page 120
ignprag= <i>option</i>	"-qignprag" on page 121
[no]info= <i>option</i>	"-qinfo" on page 124
initauto= <i>value</i>	"-qinitauto" on page 130
isolated_call= <i>names</i>	"-qisolated_call" on page 141
 C langlvl	"-qlanglvl" on page 148
[no]libansi	"-qlibansi" on page 162
[no]list	"-qlist" on page 164
[no]longlong	"-qlonglong" on page 167
[no]maxmem= <i>number</i>	"-qmaxmem" on page 172
[no]mbcs	"-qmbcs, -qdbcs" on page 173
[no]optimizeoptimize= <i>number</i>	"-O, -qoptimize" on page 179

Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent
 <i>priority=number</i>	"-qpriority (C++ only)" on page 194
proclocal, procimported, procunknown	"-qprocimported, -qproclocal, -qprocunknown (PPU only)" on page 195
 [no]proto	"-qproto (C only)" on page 197
[no]ro	"-qro (PPU only)" on page 204
[no]roconst	"-qroconst (PPU only)" on page 206
[no]showinc	"-qshowinc" on page 211
[no]source	"-qsource" on page 218
<i>spill=number</i>	"-qspill" on page 220
[no]stdinc	"-qstdinc" on page 225
[no]strict	"-qstrict" on page 227
tbtable= <i>option</i>	"-qtbtable (PPU only)" on page 238
tune= <i>option</i>	"-qtune" on page 250
[no]unrollunroll= <i>number</i>	"-qunroll" on page 252
 [no]upconv	"-qupconv (C only)" on page 255
[no]xref	"-qxref" on page 265

Usage

Most **#pragma options** directives must come before any statements in your source program; only comments, blank lines, and other pragma specifications can precede them. For example, the first few lines of your program can be a comment followed by the **#pragma options** directive:

```
/* The following is an example of a #pragma options directive: */

#pragma options langlvl=stdc89 halt=s spill=1024 source

/* The rest of the source follows ... */
```

To specify more than one compiler option with the **#pragma options** directive, separate the options using a blank space. For example:

```
#pragma options langlvl=stdc89 halt=s spill=1024 source
```

#pragma option_override

Category

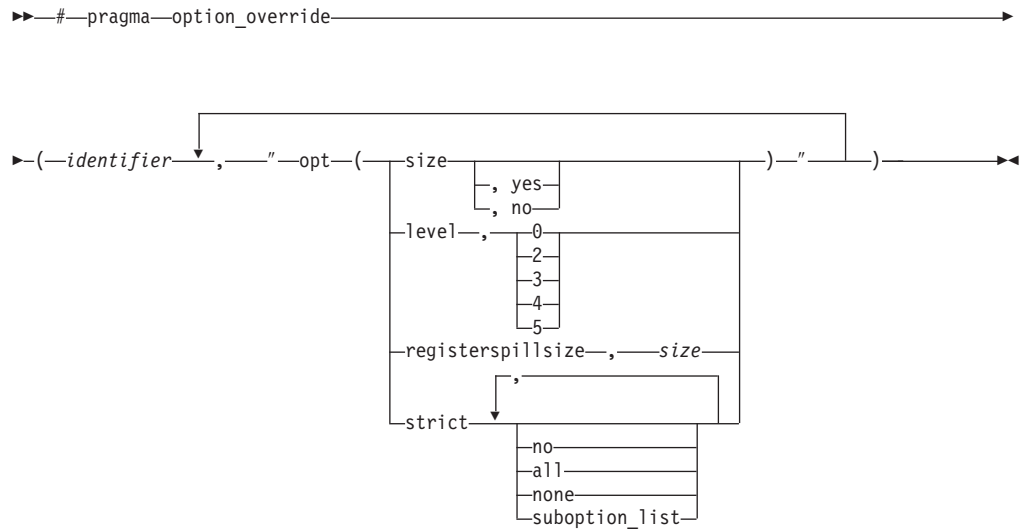
Optimization and tuning

Purpose

Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.

This enables finer control of program optimization, and can help debug errors that occur only under optimization.

Syntax



Parameters

identifier

The name of a function for which optimization options are to be overridden.

The following table shows the equivalent command line option for each pragma suboption.

#pragma option_override value	Equivalent compiler option
level, 0	-O
level, 2	-O2
level, 3	-O3
level, 4	-O4
level, 5	-O5
registerspillsize, <i>size</i>	-qspill= <i>size</i>
size	-qcompact
size, yes	
size, no	-qnocompact
strict, all	-qstrict, -qstrict=all
strict, no, none	-qnostrict
strict, <i>suboption_list</i>	-qstrict= <i>suboption_list</i>

Defaults

See the descriptions for the options listed in the table above for default settings.

Usage

The pragma takes effect only if optimization is already enabled by a command-line option. You can only specify an optimization level in the pragma *lower* than the level applied to the rest of the program being compiled.

The **#pragma option_override** directive only affects functions that are defined in the same compilation unit. The pragma directive can appear anywhere in the translation unit. That is, it can appear before or after the function definition, before or after the function declaration, before or after the function has been referenced, and inside or outside the function definition.

 This pragma cannot be used with overloaded member functions.

Examples

Suppose you compile the following code fragment containing the functions `foo` and `faa` using **-O2**. Since it contains the `#pragma option_override(faa, "opt(level, 0)")`, function `faa` will not be optimized.

```
foo(){
    .
    .
    .
}

#pragma option_override(faa, "opt(level, 0)")

faa(){
    .
    .
    .
}
```

Related information

- “-O, -qoptimize” on page 179
- “-qcompact” on page 74
- “-qspill” on page 220
- “-qstrict” on page 227

#pragma pack

Category

Object code control

Purpose

Sets the alignment of all aggregate members to a specified byte boundary.

If the byte boundary number is smaller than the natural alignment of a member, padding bytes are removed, thereby reducing the overall structure or union size.

Syntax

Default #pragma pack syntax (-qpack_semantic=ibm in effect)

```
▶▶ #pragma pack ( [nopack | number | pop] ) ▶▶
```

Defaults

Members of aggregates (structures, unions, and classes) are aligned on their natural boundaries and a structure ends on its natural boundary. The alignment of an aggregate is that of its strictest member (the member with the largest alignment requirement).

Parameters

nopack

Disables packing. Note that this parameter is not recognized when **-qpack_semantic=gnu** is in effect; a warning message is issued and the pragma is ignored.

number

is one of the following:

- 1 Aligns structure members on 1-byte boundaries, or on their natural alignment boundary, whichever is less.
- 2 Aligns structure members on 2-byte boundaries, or on their natural alignment boundary, whichever is less.
- 4 Aligns structure members on 4-byte boundaries, or on their natural alignment boundary, whichever is less.
- 8 Aligns structure members on 8-byte boundaries, or on their natural alignment boundary, whichever is less.
- 16 Aligns structure members on 16-byte boundaries, or on their natural alignment boundary, whichever is less.

pop

Sets the packing rule to that which was in effect before the current setting.

Usage

The **#pragma pack** directive applies to the definition of an aggregate type, rather than to the declaration of an instance of that type; it therefore automatically applies to all variables declared of the specified type.

The **#pragma pack** directive modifies the current alignment rule for only the members of structures whose declarations follow the directive. It does not affect the alignment of the structure directly, but by affecting the alignment of the members of the structure, it may affect the alignment of the overall structure.

The **#pragma pack** directive cannot increase the alignment of a member, but rather can decrease the alignment. For example, for a member with data type of short, a **#pragma pack(1)** directive would cause that member to be packed in the structure on a 1-byte boundary, while a **#pragma pack(4)** directive would have no effect.

The **#pragma pack** directive causes bit fields to cross bit field container boundaries.

```
#pragma pack(2)
struct A{
  int a:31;
  int b:2;
}x;

int main(){
  printf("size of S = %d\n", sizeof(s));
}
```

When compiled and run, the output is:
size of S = 6

But if you remove the `#pragma pack` directive, you get this output:
size of S = 8

The **#pragma pack** directive applies only to complete declarations of structures or unions; this excludes forward declarations, in which member lists are not specified. For example, in the following code fragment, the alignment for struct S is 4, since this is the rule in effect when the member list is declared:

```
#pragma pack(1)
struct S;
#pragma pack(4)
struct S { int i, j, k; };
```

A nested structure has the alignment that precedes its declaration, not the alignment of the structure in which it is contained, as shown in the following example:

```
#pragma pack (4)                // 4-byte alignment
    struct nested {
        int x;
        char y;
        int z;
    };

    #pragma pack(1)              // 1-byte alignment
    struct packedcxx{
        char a;
        short b;
        struct nested s1;       // 4-byte alignment
    };
```

If more than one **#pragma pack** directive appears in a structure defined in an inlined function, the **#pragma pack** directive in effect at the beginning of the structure takes precedence.

Examples

The following example shows how the **#pragma pack** directive can be used to set the alignment of a structure definition:

```
// header file file.h

#pragma pack(1)

struct jeff{                // this structure is packed
    short bill;             // along 1-byte boundaries
    int *chris;
};
#pragma pack(pop)           // reset to previous alignment rule

// source file anyfile.c

#include "file.h"

struct jeff j;              // uses the alignment specified
                           // by the pragma pack directive
                           // in the header file and is
                           // packed along 1-byte boundaries
```

This example shows how a **#pragma pack** directive can affect the size and mapping of a structure:

```

struct s_t {
    char a;
    int b;
    short c;
    int d;
}S;

```

Default mapping:

size of s_t = 16
 offset of a = 0
 offset of b = 4
 offset of c = 8
 offset of d = 12
 alignment of a = 1
 alignment of b = 4
 alignment of c = 2
 alignment of d = 4

With #pragma pack(1):

size of s_t = 11
 offset of a = 0
 offset of b = 1
 offset of c = 5
 offset of d = 7
 alignment of a = 1
 alignment of b = 1
 alignment of c = 1
 alignment of d = 1

The following example defines a union uu containing a structure as one of its members, and declares an array of 2 unions of type uu:

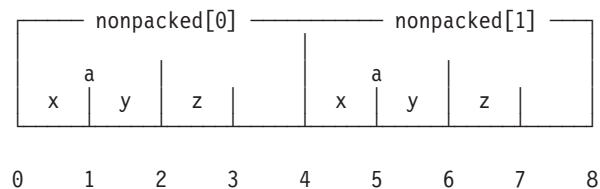
```

union uu {
    short a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu nonpacked[2];

```

Since the largest alignment requirement among the union members is that of short a, namely, 2 bytes, one byte of padding is added at the end of each union in the array to enforce this requirement:



The next example uses **#pragma pack(1)** to set the alignment of unions of type uu to 1 byte:

```

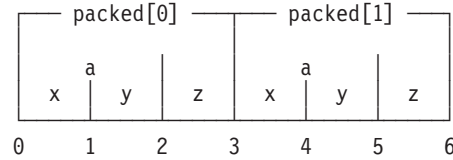
#pragma pack(1)

union uu {
    short a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu pack_array[2];

```

Now, each union in the array packed has a length of only 3 bytes, as opposed to the 4 bytes of the previous case:



Related information

- “-qalign” on page 53
- “Using alignment modifiers” in the *XL C/C++ Optimization and Programming Guide*

#pragma priority (C++ only)

See “-qpriority (C++ only)” on page 194.

#pragma reachable

Category

Optimization and tuning

Purpose

Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.

By informing the compiler that the instruction after the specified function can be reached from a point in your program other than the return statement in the named function, the pragma allows for additional opportunities for optimization.

Note: The compiler automatically inserts **#pragma reachable** directives for the setjmp family of functions (setjmp, _setjmp, sigsetjmp, and _sigsetjmp) when you include the setjmp.h header file.

Syntax

```

▶▶ #pragma reachable ( function_name ) ▶▶

```

Parameters

function_name

The name of a function preceding the instruction which is reachable from a point in the program other than the function’s return statement.

Defaults

Not applicable.

Related information

- “#pragma leaves” on page 288

#pragma reg_killed_by

Category

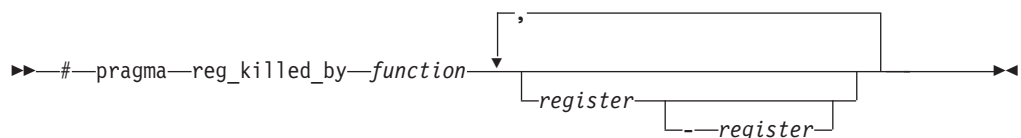
Optimization and tuning

Purpose

Specifies registers that may be altered by functions specified by **#pragma mc_func**.

Ordinarily, code generated for functions specified by **#pragma mc_func** may alter any or all volatile registers available on your system. You can use **#pragma reg_killed_by** to explicitly list a specific set of volatile registers to be altered by such functions. Registers not in this list will not be altered.

Syntax



Parameters

function

The name of a function previously defined using the **#pragma mc_func** directive.

register

The symbolic name(s) of either a single register or a range of registers to be altered by the named *function*. The symbolic name must be a valid register name on the target platform. Valid registers for the PPU are:

cr0, cr1, and cr5 to cr7

Condition registers

ctr

Count register

gr0 and gr3 to gr12

General purpose registers

fp0 to fp13

Floating-point registers

fs

Floating point and status control register

lr

Link register

vr0 to vr31

Vector registers

xer

Fixed-point exception register

Valid registers for the SPU are 0 to 127, or v0 to v127.

You can identify a range of registers by providing the symbolic names of both starting and ending registers, separated by a dash.

If no *register* is specified, no volatile registers will be killed by the named *function*.

Examples

The following example shows how to use **#pragma reg_killed_by** to list a specific set of volatile registers to be used by the function defined by **#pragma mc_func**.

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
/*      addc      r3 <- r3, r4      */
/*      addze     r3 <- r3, carry bit */

#pragma reg_killed_by add_logical gr3, xer
/* only gpr3 and the xer are altered by this function */

main() {
    int i,j,k;

    i = 4;
    k = -4;
    j = add_logical(i,k);
    printf("\n\nresult = %d\n\n",j);
}
```

Related information

- “#pragma mc_func” on page 293

#pragma report (C++ only)

Category

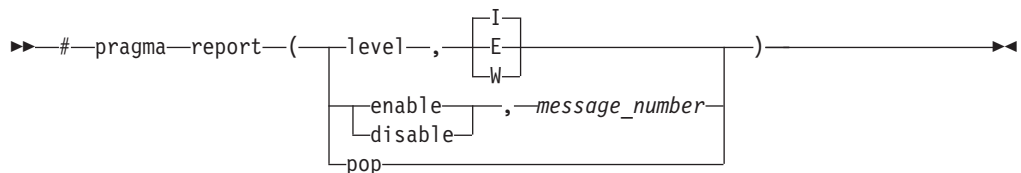
Listings, messages and compiler information

Purpose

Controls the generation of diagnostic messages.

The pragma allows you to specify a minimum severity level for a message for it to display, or allows you to enable or disable a specific message regardless of the prevailing report level.

Syntax



Defaults

The default report level is Informational (I), which displays messages of all types.

Parameters

level

Indicates that the pragma is set according to the minimum severity level of diagnostic messages to display.

- E** Indicates that only error messages will display. Error messages are of the highest severity. This is equivalent to the **-qflag=e:e** compiler option.
- W** Indicates that warning and error messages will display. This is equivalent to the **-qflag=w:w** compiler option.
- I** Indicates that all diagnostic messages will display: warning, error and informational messages. Informational messages are of the lowest severity. This is equivalent to the **-qflag=i:i** compiler option.

enable

Enables the specified *message_number*.

disable

Disables the specified *message_number*.

message_number

Represents a message identifier, which consists of a prefix followed by the message number; for example, CCN1004.

pop

Reverts the report level to that which was previously in effect. If no previous report level has been specified, a warning is issued, and the report level remains unchanged.

Usage

The pragma takes precedence over **#pragma info** and most compiler options. For example, if you use **#pragma report** to disable a compiler message, that message will not be displayed with any **-qflag** compiler option setting.

Related information

- “-qflag” on page 97

#pragma STDC cx_limited_range

Category

Optimization and tuning

Purpose

Instructs the compiler that complex division and absolute value are only invoked with values such that intermediate calculation will not overflow or lose significance.

Syntax

```

▶▶ #pragma STDC cx_limited_range { off | on | default }

```

Usage

Using values outside the limited range may generate wrong results, where the limited range is defined such that the “obvious symbolic definition” will not overflow or run out of precision.

The pragma is effective from its first occurrence until another **cx_limited_range** pragma is encountered, or until the end of the translation unit. When the pragma occurs inside a compound statement (including within a nested compound statement), it is effective from its first occurrence until another **cx_limited_range** pragma is encountered, or until the end of the compound statement.

Examples

The following example shows the use of the pragma for complex division:

```
#include <complex.h>

_Complex double a, b, c, d;
void p() {

    d = b/c;

    {

        #pragma STDC CX_LIMITED_RANGE ON

        a = b / c;

    }

}
```

The following example shows the use of the pragma for complex absolute value:

```
#include <complex.h>

_Complex double cd = 10.10 + 10.10*I;
int p() {

    #pragma STDC CX_LIMITED_RANGE ON

    double d = cabs(cd);

}
```

Related information

- "Standard pragmas" in the *XL C/C++ Language Reference*

#pragma stream_unroll

Category

Optimization and tuning

Purpose

When optimization is enabled, breaks a stream contained in a for loop into multiple streams.


Syntax

```
▶▶ #pragma stream_unroll [ (—number—) ] ▶▶
```

Parameters

number

A loop unrolling factor.  The value of *number* is a positive integral



constant expression.  The value of *number* is a positive scalar integer or compile-time constant initialization expression.

An unroll factor of 1 disables unrolling.

If *number* is not specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Usage

To enable stream unrolling, you must specify **-qhot** and **-qstrict**, or use optimization level **-O4** or higher. If **-qstrict** is in effect, no stream unrolling takes place.

For stream unrolling to occur, the **#pragma stream_unroll** directive must be the last pragma specified preceding a for loop.  Specifying **#pragma stream_unroll** more than once for the same for loop or combining it with other loop unrolling pragmas (**#pragma unroll**, **#pragma nounroll**, **#pragma unrollandfuse**, **#pragma nounrollandfuse**) results in a warning.  The compiler silently ignores all but the last of multiple loop unrolling pragmas specified on the same for loop.

Examples

The following is an example of how **#pragma stream_unroll** can increase performance.

```
int i, m, n;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];

....

#pragma stream_unroll(4)
for (i=1; i<n; i++) {
    a[i] = b[i] * c[i];
}
```

The unroll factor of 4 reduces the number of iterations from *n* to *n*/4, as follows:

```
for (i=1; i<n/4; i++) {
    a[i] = b[i] + c[i];
    a[i+m] = b[i+m] + c[i+m];
    a[i+2*m] = b[i+2*m] + c[i+2*m];
    a[i+3*m] = b[i+3*m] + c[i+3*m];
}
```

Related information

- “-qunroll” on page 252
- “#pragma unrollandfuse” on page 308

#pragma strings

See “-qro (PPU only)” on page 204.

#pragma unroll

See “-qunroll” on page 252.


```

int i, j;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];

....

#pragma unrollandfuse(2)
for (i=1; i<1000; i++) {
    for (j=1; j<1000; j++) {
        a[j][i] = b[i][j] * c[j][i];
    }
}

```

The for loop below shows a possible result of applying the **#pragma unrollandfuse(2)** directive to the loop shown above:

```

for (i=1; i<1000; i=i+2) {
    for (j=1; j<1000; j++) {
        a[j][i] = b[i][j] * c[j][i];
        a[j][i+1] = b[i+1][j] * c[j][i+1];
    }
}

```

You can also specify multiple **#pragma unrollandfuse** directives in a nested loop structure.

```

int i, j, k;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];
int d[1000][1000];
int e[1000][1000];

....

#pragma unrollandfuse(4)
for (i=1; i<1000; i++) {
    #pragma unrollandfuse(2)
    for (j=1; j<1000; j++) {
        for (k=1; k<1000; k++) {
            a[j][i] = b[i][j] * c[j][i] + d[j][k] * e[i][k];
        }
    }
}

```

Related information

- “-qunroll” on page 252
- “#pragma stream_unroll” on page 306

#pragma weak

Category

Object code control

Purpose

Prevents the linker from issuing error messages if it encounters a symbol multiply-defined during linking, or if it does not find a definition for a symbol.

The pragma can be used to allow a program to call a user-defined function that has the same name as a library function. By marking the library function definition as "weak", the programmer can reference a "strong" version of the function and cause the linker to accept multiple definitions of a global symbol in the object code. While this pragma is intended for use primarily with functions, it will also work for most data objects.

Syntax

```
▶ #pragma weak name1 [=name2]
```

Parameters

name1

A name of a data object or function with external linkage.

name2

A name of a data object or function with external linkage.

▶ **C++** *name2* must not be a member function. If *name2* is a template function, you must explicitly instantiate the template function.

▶ **C++** Names must be specified using their mangled names. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file. (See also "Name mangling" in the *XL C/C++ Language Reference* for details on using the extern "C" linkage specifier on declarations to prevent name mangling.)

Usage

There are two forms of the **weak** pragma:

#pragma weak *name1*

This form of the pragma marks the definition of the *name1* as "weak" in a given compilation unit. If *name1* is referenced from anywhere in the program, the linker will use the "strong" version of the definition (that is, the definition not marked with **#pragma weak**), if there is one. If there is no strong definition, the linker will use the weak definition; if there are multiple weak definitions, it is unspecified which weak definition the linker will select (typically, it uses the definition found in the first object file specified on the command line during the link step). *name1* must be defined in the same compilation unit as **#pragma weak**. If *name1* is referenced, but no definition of it can be found, it is assigned a value of 0.

#pragma weak *name1*=*name2*

This form of the pragma creates a weak definition of the *name1* for a given compilation unit, and an alias for *name2*. If *name1* is referenced from anywhere in the program, the linker will use the "strong" version of the definition (that is, the definition not marked with **#pragma weak**), if there is one. If there is no strong definition, the linker will use the weak definition, which resolves to the definition of *name2*. If there are multiple weak definitions, it is unspecified which weak definition the linker will select (typically, it uses the definition found in the first object file specified on the command line during the link step).

name2 must be defined in the same compilation unit as **#pragma weak**. *name1* may or may not be declared in the same compilation unit as the **#pragma weak**, but must never be defined in the compilation unit. If *name1* is declared in the compilation unit, *name1*'s declaration must be compatible to that of *name2*. For example, if *name2* is a function, *name1* must have the same return and argument types as *name2*.

This pragma should not be used with uninitialized global data, or with shared library data objects that are exported to executables.

Examples

The following is an example of the **#pragma weak name1** form:

```
// Compilation unit 1:

#include <stdio.h>

void foo();

int main()
{
    foo();
}

// Compilation unit 2:

#include <stdio.h>

#if __cplusplus
#pragma weak
#else
#pragma weak foo
#endif
void foo()
{
    printf("Foo called from compilation unit 2\n");
}

// Compilation unit 3:

#include <stdio.h>

void foo()
{
    printf("Foo called from compilation unit 3\n");
}
```

If all three compilation units are compiled and linked together, the linker will use the strong definition of `foo` in compilation unit 3 for the call to `foo` in compilation unit 1, and the output will be:

Foo called from compilation unit 3

If only compilation unit 1 and 2 are compiled and linked together, the linker will use the weak definition of `foo` in compilation unit 2, and the output will be:

Foo called from compilation unit 2

The following is an example of the **#pragma weak name1=name2** form:

```
// Compilation unit 1:

#include <stdio.h>
```

```

void foo();

int main()
{
    foo();
}

// Compilation unit 2:

#include <stdio.h>

void foo(); // optional

#ifdef __cplusplus
#pragma weak
#else
#pragma weak foo = foo2
#endif
void foo2()
{
    printf("Hello from foo2!\n");
}

// Compilation unit 3:

#include <stdio.h>

void foo()
{
    printf("Hello from foo!\n");
}

```

If all three compilation units are compiled and linked together, the linker will use the strong definition of `foo` in compilation unit 3 for the call to `foo` from compilation unit 1, and the output will be:

```
Hello from foo!
```

If only compilation unit 1 and 2 are compiled and linked together, the linker will use the weak definition of `foo` in compilation unit 2, which is an alias for `foo2`, and the output will be:

```
Hello from foo2!
```

Related information

- "The weak variable attribute" in the *XL C/C++ Language Reference*
- "The weak function attribute" in the *XL C/C++ Language Reference*
- "#pragma map" on page 290

Pragma directives for parallel processing

Parallel processing operations are controlled by pragma directives in your program source. The pragmas have effect only when parallelization is enabled with the **-qsmp** compiler option. If you use OpenMP directives to create the SPU sections of your application, you must compile using one of the **cbe** invocation commands, such as **cbexlc**.

#pragma omp atomic

Purpose

The **omp atomic** directive identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.

Syntax

```

▶▶ #pragma omp atomic [statement]

```

where *statement* is an expression statement of scalar type that takes one of the forms that follow:

<i>statement</i>	Conditions
<code>x bin_op = expr</code>	where: <i>bin_op</i> is one of: <code> + * - / & ^ << >></code> <i>expr</i> is an expression of scalar type that does not reference <i>x</i> .
<code>x++</code>	
<code>++x</code>	
<code>x--</code>	
<code>--x</code>	

Usage

Load and store operations are atomic only for object *x*. Evaluation of *expr* is not atomic.

All atomic references to a given object in your program must have a compatible type.

Objects that can be updated in parallel and may be subject to race conditions should be protected with the **omp atomic** directive.

Examples

```

extern float x[], *p = x, y;
/* Protect against race conditions among multiple updates. */
#pragma omp atomic
x[index[i]] += y;
/* Protect against races with updates through x. */
#pragma omp atomic
p[i] -= 1.0f;

```

#pragma omp parallel

Purpose

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen block of code.

Syntax

```

▶▶ #pragma omp parallel [clause]

```

Parameters

clause is any of the following:

if (*exp*)

When the **if** argument is specified, the program code executes in parallel only if the scalar expression represented by *exp* evaluates to a nonzero value at run time. Only one **if** clause can be specified.

private (*list*)

Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list*)

Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

num_threads (*int_exp*)

The value of *int_exp* is an integer expression that specifies the number of threads to use for the parallel region. If dynamic adjustment of the number of threads is also enabled, then *int_exp* specifies the maximum number of threads to be used.

shared (*list*)

Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

default (**shared** | **none**)

Defines the default data scope of variables in each thread. Only one **default** clause can be specified on an **omp parallel** directive.

Specifying **default(shared)** is equivalent to stating each variable in a **shared(list)** clause.

Specifying **default(none)** requires that each data variable visible to the parallelized statement block must be explicitly listed in a data scope clause, with the exception of those variables that are:

- const-qualified,
- specified in an enclosed data scope attribute clause, or,
- used as a loop control variable referenced only by a corresponding **omp for** or **omp parallel for** directive.

copyin (*list*)

For each data variable specified in *list*, the value of the data variable in the master thread is copied to the thread-private copies at the beginning of the parallel region. Data variables in *list* are separated by commas.

Each data variable specified in the **copyin** clause must be a **threadprivate** variable.

reduction (*operator: list*)

Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

Variables specified in the **reduction** clause:

- Must be of a type appropriate to the operator.
- Must be shared in the enclosing context.
- Must not be const-qualified.
- Must not have pointer type.

Usage

When a parallel region is encountered, a logical team of threads is formed. Each thread in the team executes all statements within a parallel region except for work-sharing constructs. Work within work-sharing constructs is distributed among the threads in a team.

Loop iterations must be independent before the loop can be parallelized. An implied barrier exists at the end of a parallelized statement block.

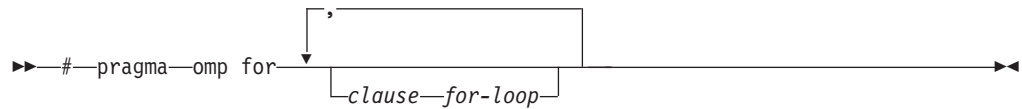
Nested parallel regions are always serialized.

#pragma omp for

Purpose

The **omp for** directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.

Syntax



Parameters

clause is any of the following:

private (*list*)

Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list*)

Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

lastprivate (*list*)

Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last iteration. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.

reduction (*operator:list*)

Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

Variables specified in the **reduction** clause:

- must be of a type appropriate to the operator.
- must be shared in the enclosing context.
- must not be const-qualified.
- must not have pointer type.

ordered

Specify this clause if an ordered construct is present within the dynamic extent of the **omp for** directive.

schedule (*type*)

Specifies how iterations of the **for** loop are divided among available threads. Acceptable values for *type* are:

auto With **auto**, scheduling is delegated to the compiler and runtime system. The compiler and runtime system can choose any possible mapping of iterations to threads (including all possible valid schedules) and these may be different in different loops.

dynamic

Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*).

Chunks are dynamically assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

dynamic,*n*

As above, except chunks are set to size *n*. *n* must be an integral assignment expression of value 1 or greater.

guided

Chunks are made progressively smaller until the default minimum chunk size is reached. The first chunk is of size **ceiling**(*number_of_iterations*/*number_of_threads*). Remaining chunks are of size **ceiling**(*number_of_iterations_left*/*number_of_threads*).

The minimum chunk size is 1.

Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

guided,*n*

As above, except the minimum chunk size is set to *n*. *n* must be an integral assignment expression of value 1 or greater.

runtime

Scheduling policy is determined at run time. Use the OMP_SCHEDULE environment variable to set the scheduling type and chunk size.

static

Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*). Each thread is assigned a separate chunk.

This scheduling policy is also known as *block scheduling*.

static,*n*

Iterations of a loop are divided into chunks of size *n*. Each chunk is assigned to a thread in *round-robin* fashion.

n must be an integral assignment expression of value 1 or greater.

This scheduling policy is also known as *block cyclic scheduling*.

Note: if $n=1$, iterations of a loop are divided into chunks of size 1 and each chunk is assigned to a thread in *round-robin* fashion. This scheduling policy is also known as *block cyclic scheduling*

nowait

Use this clause to avoid the implied **barrier** at the end of the **for** directive. This is useful if you have multiple independent work-sharing sections or iterative loops within a given parallel region. Only one **nowait** clause can appear on a given **for** directive.

and where *for_loop* is a for loop construct with the following canonical shape:

```
for (init_expr; exit_cond; incr_expr)
  statement
```

where:

<i>init_expr</i>	takes form:	<i>iv</i> = <i>b</i> <i>integer-type iv</i> = <i>b</i>
<i>exit_cond</i>	takes form:	<i>iv</i> <= <i>ub</i> <i>iv</i> < <i>ub</i> <i>iv</i> >= <i>ub</i> <i>iv</i> > <i>ub</i>
<i>incr_expr</i>	takes form:	++ <i>iv</i> <i>iv</i> ++ -- <i>iv</i> <i>iv</i> -- <i>iv</i> += <i>incr</i> <i>iv</i> -= <i>incr</i> <i>iv</i> = <i>iv</i> + <i>incr</i> <i>iv</i> = <i>incr</i> + <i>iv</i> <i>iv</i> = <i>iv</i> - <i>incr</i>

and where:

<i>iv</i>	Iteration variable. The iteration variable must be a signed integer not modified anywhere within the for loop. It is implicitly made private for the duration of the for operation. If not specified as lastprivate , the iteration variable will have an indeterminate value after the operation completes.
<i>b</i> , <i>ub</i> , <i>incr</i>	Loop invariant signed integer expressions. No synchronization is performed when evaluating these expressions and evaluated side effects may result in indeterminate values.

Usage

This pragma must appear immediately before the loop or loop block directive to be affected.

Program sections using the **omp for** pragma must be able to produce a correct result regardless of which thread executes a particular iteration. Similarly, program correctness must not rely on using a particular scheduling algorithm.

The for loop iteration variable is implicitly made private in scope for the duration of loop execution. This variable must not be modified within the body of the for loop. The value of the increment variable is indeterminate unless the variable is specified as having a data scope of **lastprivate**.

An implicit barrier exists at the end of the for loop unless the **nowait** clause is specified.

Restrictions are:

- The for loop must be a structured block, and must not be terminated by a break statement.
- Values of the loop control expressions must be the same for all iterations of the loop.
- An **omp for** directive can accept only one **schedule** clauses.
- The value of *n* (chunk size) must be the same for all threads of a parallel region.

#pragma omp ordered

Purpose

The **omp ordered** directive identifies a structured block of code that must be executed in sequential order.

Syntax

►► #pragma omp ordered _____ ◀◀

Usage

The **omp ordered** directive must be used as follows:

- It must appear within the extent of a **omp for** or **omp parallel for** construct containing an **ordered** clause.
- It applies to the statement block immediately following it. Statements in that block are executed in the same order in which iterations are executed in a sequential loop.
- An iteration of a loop must not execute the same **omp ordered** directive more than once.
- An iteration of a loop must not execute more than one distinct **omp ordered** directive.

#pragma omp parallel for

Purpose

The **omp parallel for** directive effectively combines the **omp parallel** and **omp for** directives. This directive lets you define a parallel region containing a single **for** directive in one step.

Syntax

►► #pragma omp for _____ ◀◀

┌ clause-for-loop ───────────┐

Usage

With the exception of the **nowait** clause, clauses and restrictions described in the **omp parallel** and **omp for** directives also apply to the **omp parallel for** directive.

#pragma omp section, #pragma omp sections

The **omp sections** directive distributes work among threads bound to a defined parallel region.

Syntax



Parameters

clause is any of the following:

private (*list*)

Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list*)

Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

lastprivate (*list*)

Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last **section**. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.

reduction (*operator: list*)

Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

Variables specified in the **reduction** clause:

- must be of a type appropriate to the operator.
- must be shared in the enclosing context.
- must not be const-qualified.
- must not have pointer type.

nowait

Use this clause to avoid the implied **barrier** at the end of the **sections** directive. This is useful if you have multiple independent work-sharing sections within a given parallel region. Only one **nowait** clause can appear on a given **sections** directive.

Usage

The **omp section** directive is optional for the first program code segment inside the **omp sections** directive. Following segments must be preceded by an **omp section** directive. All **omp section** directives must appear within the lexical construct of the program source code segment associated with the **omp sections** directive.

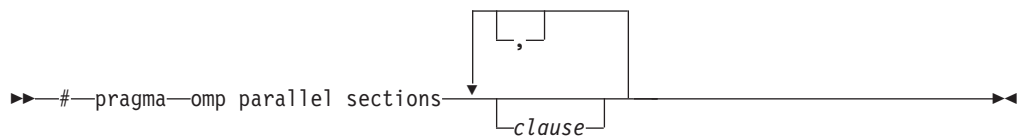
When program execution reaches a **omp sections** directive, program segments defined by the following **omp section** directive are distributed for parallel execution among available threads. A barrier is implicitly defined at the end of the larger program region associated with the **omp sections** directive unless the **nowait** clause is specified.

#pragma omp parallel sections

Purpose

The **omp parallel sections** directive effectively combines the **omp parallel** and **omp sections** directives. This directive lets you define a parallel region containing a single **sections** directive in one step.

Syntax



Usage

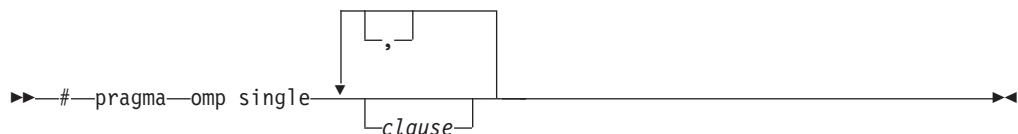
All clauses and restrictions described in the **omp parallel** and **omp sections** directives apply to the **omp parallel sections** directive.

#pragma omp single

Purpose

The **omp single** directive identifies a section of code that must be run by a single available thread.

Syntax



Parameters

clause is any of the following:

private (*list*)

Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

A variable in the **private** clause must not also appear in a **copyprivate** clause for the same **omp single** directive.

copyprivate (*list*)

Broadcasts the values of variables specified in *list* from one member of the team to other members. This occurs after the execution of the structured block associated with the **omp single** directive, and before any of the threads leave the barrier at the end of the construct. For all other threads in the team, each variable in the *list* becomes defined with the value of the corresponding variable in the thread that executed the structured block. Data variables in *list* are separated by commas. Usage restrictions for this clause are:

- A variable in the **copyprivate** clause must not also appear in a **private** or **firstprivate** clause for the same **omp single** directive.
- If an **omp single** directive with a **copyprivate** clause is encountered in the dynamic extent of a parallel region, all variables specified in the **copyprivate** clause must be private in the enclosing context.
- Variables specified in **copyprivate** clause within dynamic extent of a parallel region must be private in the enclosing context.
- A variable that is specified in the **copyprivate** clause must have an accessible and unambiguous copy assignment operator.
- The **copyprivate** clause must not be used together with the **nowait** clause.

firstprivate (*list*)

Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

A variable in the **firstprivate** clause must not also appear in a **copyprivate** clause for the same **omp single** directive.

nowait

Use this clause to avoid the implied **barrier** at the end of the **single** directive. Only one **nowait** clause can appear on a given **single** directive. The **nowait** clause must not be used together with the **copyprivate** clause.

Usage

An implied barrier exists at the end of a parallelized statement block unless the **nowait** clause is specified.

#pragma omp master

Purpose

The **omp master** directive identifies a section of code that must be run only by the master thread.

Syntax

►—#—pragma—omp master—◄

Usage

Threads other than the master thread will not execute the statement block associated with this construct.

No implied barrier exists on either entry to or exit from the master section.

#pragma omp critical

Purpose

The **omp critical** directive identifies a section of code that must be executed by a single thread at a time.

Syntax

```
▶▶ #pragma omp critical (name) ▶▶
```

where *name* can optionally be used to identify the critical region. Identifiers naming a critical region have external linkage and occupy a namespace distinct from that used by ordinary identifiers.

Usage

A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name. Critical sections not specifically named by **omp critical** directive invocation are mapped to the same unspecified name.

#pragma omp barrier

Purpose

The **omp barrier** directive identifies a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point. Statement execution past the **omp barrier** point then continues in parallel.

Syntax

```
▶▶ #pragma omp barrier ▶▶
```

Usage

The **omp barrier** directive must appear within a block or compound statement. For example:

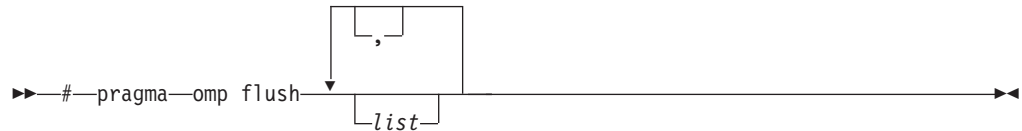
```
if (x!=0) {  
    #pragma omp barrier    /* valid usage    */  
}  
if (x!=0)  
    #pragma omp barrier    /* invalid usage */
```

#pragma omp flush

Purpose

The **omp flush** directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.

Syntax



where *list* is a comma-separated list of variables that will be synchronized.

Usage

If *list* includes a pointer, the pointer is flushed, not the object being referred to by the pointer. If *list* is not specified, all shared objects are synchronized except those inaccessible with automatic storage duration.

An implied **flush** directive appears in conjunction with the following directives:

- **omp barrier**
- Entry to and exit from **omp critical**.
- Exit from **omp parallel**.
- Exit from **omp for**.
- Exit from **omp sections**.
- Exit from **omp single**.

The **omp flush** directive must appear within a block or compound statement. For example:

```

if (x!=0) {
    #pragma omp flush    /* valid usage    */
}
if (x!=0)
    #pragma omp flush    /* invalid usage */

```

#pragma omp threadprivate

Purpose

The **omp threadprivate** directive makes the named file-scope, namespace-scope, or static block-scope variables private to a thread.

Syntax



where *identifier* is a file-scope, name space-scope or static block-scope variable.

Usage

Each copy of an **omp threadprivate** data variable is initialized once prior to first use of that copy. If an object is changed before being used to initialize a **threadprivate** data variable, behavior is unspecified.

A thread must not reference another thread's copy of an **omp threadprivate** data variable. References will always be to the master thread's copy of the data variable when executing serial and master regions of the program.

Use of the **omp threadprivate** directive is governed by the following points:

- An **omp threadprivate** directive must appear at file scope outside of any definition or declaration.
- The **omp threadprivate** directive is applicable to static-block scope variables and may appear in lexical blocks to reference those block-scope variables. The directive must appear in the scope of the variable and not in a nested scope, and must precede all references to variables in its list.
- A data variable must be declared with file scope prior to inclusion in an **omp threadprivate** directive *list*.
- An **omp threadprivate** directive and its *list* must lexically precede any reference to a data variable found in that *list*.
- A data variable specified in an **omp threadprivate** directive in one translation unit must also be specified as such in all other translation units in which it is declared.
- Data variables specified in an **omp threadprivate** *list* must not appear in any clause other than the **copyin**, **copyprivate**, **if**, **num_threads**, and **schedule** clauses.
- The address of a data variable in an **omp threadprivate** *list* is not an address constant.
- A data variable specified in an **omp threadprivate** *list* must not have an incomplete or reference type.

Chapter 5. Compiler predefined macros

Predefined macros can be used to conditionally compile code for specific compilers, specific versions of compilers, specific environments and/or specific language features.

Predefined macros fall into several categories:

- “General macros”
- “Macros related to the platform” on page 327
- “Macros related to compiler features” on page 328

“Examples of predefined macros” on page 335 show how you can use them in your code.

General macros

The following predefined macros are always predefined by the compiler. Unless noted otherwise, all the following macros are *protected*, which means that the compiler will issue a warning if you try to undefine or redefine them.

Table 34. General predefined macros

Predefined macro name	Description	Predefined value
__BASE_FILE__	Indicates the name of the primary source file.	The fully qualified file name of the primary source file.
__DATE__	Indicates the date that the source file was preprocessed.	A character string containing the date when the source file was preprocessed.
__FILE__	Indicates the name of the preprocessed source file.	A character string containing the name of the preprocessed source file.
__FUNCTION__	Indicates the name of the function currently being compiled.	A character string containing the name of the function currently being compiled.
__LINE__	Indicates the current line number in the source file.	An integer constant containing the line number in the source file.
__SIZE_TYPE__	Indicates the underlying type of <code>size_t</code> on the current platform. Not protected.	For PPU, unsigned int in 32-bit compilation mode and unsigned long in 64-bit compilation mode. For SPU, unsigned long.
__TIME__	Indicates the time that the source file was preprocessed.	A character string containing the time when the source file was preprocessed.

Table 34. General predefined macros (continued)

Predefined macro name	Description	Predefined value
<code>__TIMESTAMP__</code>	Indicates the date and time when the source file was last modified. The value changes as the compiler processes any include files that are part of your source program.	<p>A character string literal in the form "<i>Day Mmm dd hh:mm:ss yyyy</i>", where::</p> <p><i>Day</i> Represents the day of the week (Mon, Tue, Wed, Thu, Fri, Sat, or Sun).</p> <p><i>Mmm</i> Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).</p> <p><i>dd</i> Represents the day. If the day is less than 10, the first d is a blank character.</p> <p><i>hh</i> Represents the hour.</p> <p><i>mm</i> Represents the minutes.</p> <p><i>ss</i> Represents the seconds.</p> <p><i>yyyy</i> Represents the year.</p>

Macros indicating the XL C/C++ compiler product

Macros related to the XL C/C++ compiler are always predefined, and are protected (the compiler will issue a warning if you try to undefine or redefine them).

Table 35. Compiler product predefined macros




Predefined macro name	Description	Predefined value
 <code>__IBMC__</code>	Indicates the level of the XL C compiler.	<p>An integer in the format <i>VRM</i>, where :</p> <p><i>V</i> Represents the version number</p> <p><i>R</i> Represents the release number</p> <p><i>M</i> Represents the modification number</p> <p>In XL C/C++ V10.1, the value of the macro is 1010.</p>
 <code>__IBMCPP__</code>	Indicates the level of the XL C++ compiler.	<p>An integer in the format <i>VRM</i>, where :</p> <p><i>V</i> Represents the version number</p> <p><i>R</i> Represents the release number</p> <p><i>M</i> Represents the modification number</p> <p>In XL C/C++ V10.1, the value of the macro is 1010.</p>
 <code>__xlc__</code>	Indicates the level of the XL C compiler.	<p>A string in the format "<i>V.R.M.F</i>", where:</p> <p><i>V</i> Represents the version number</p> <p><i>R</i> Represents the release number</p> <p><i>M</i> Represents the modification number</p> <p><i>F</i> Represents the fix level</p> <p>In XL C/C++ V10.1, the value of the macro is "10.1.0.0".</p>

Table 35. Compiler product predefined macros (continued)

Predefined macro name	Description	Predefined value
<code>__xLC__</code>	Indicates the level of the XL C++ compiler. Using the XL C compiler also automatically defines this macro.	A four-digit hexadecimal integer in the format 0xVRRR, where: <div style="margin-left: 20px;">V Represents the version number</div> <div style="margin-left: 20px;">R Represents the release number</div> In XL C/C++ V10.1, the value of the macro is 0x0a01.

Macros related to the platform

The following predefined macros are provided to facilitate porting applications between platforms. All platform-related predefined macros are unprotected and may be undefined or redefined without warning unless otherwise specified.

Table 36. Platform-related predefined macros


Predefined macro name	Description	Predefined value	Predefined under the following conditions
<code>_BIG_ENDIAN,</code> <code>__BIG_ENDIAN__</code>	Indicates that the platform is big-endian (that is, the most significant byte is stored at the memory location with the lowest address).	1	Always predefined.
<code>__ELF__</code>	Indicates that the ELF object model is in effect.	1	Always predefined for the Linux platform.
 <code>__GXX_WEAK__</code>	Indicates that weak symbols are supported (used for template instantiation by the linker).	1	Always predefined.
<code>_ILP32,</code> <code>__ILP32__</code>	Indicates that the target platform uses 32-bit int, long int, and pointer types.	1	Predefined when the target platform uses 32-bit int, long int, and pointer types.
<code>_LP32,</code> <code>__LP32__</code>	Indicates that the target platform uses 32 bits for int, long int, and pointer types.	1	Predefined when the target platform uses 32-bit int, long int, and pointer types.
<code>_LP64,</code> <code>__LP64__</code>	Indicates that the target platform uses 64-bit long int and pointer types, and a 32-bit int type.	1	Predefined when the target platform uses 64-bit long int and pointer types, and 32-bit a int type.
<code>__powerpc,</code> <code>__powerpc__</code>	Indicates that the target is a Power architecture.	1	Predefined when compilation targets the PPU.
<code>__powerpc64__</code>	Indicates that the target is a Power architecture and that 64-bit compilation mode is enabled.	1	Predefined when compilation targets the PPU and -q64 is in effect.
<code>__PPC,</code> <code>__PPC__</code>	Indicates that the target is a Power architecture.	1	Predefined when compilation targets the PPU.
<code>__PPC64__</code>	Indicates that the target is a Power architecture and that 64-bit compilation mode is enabled.	1	Predefined when compilation targets the PPU and -q64 is in effect.
<code>__THW_CBEA__</code>	Indicates that the target architecture is Cell Broadband Engine.	1	Always predefined.

Table 36. Platform-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined under the following conditions
<code>__unix</code> , <code>__unix__</code>	Indicates that the operating system is a variety of UNIX.	1	Predefined for a UNIX-like platform targeting PPU.

Macros related to compiler features

Feature-related macros are predefined according to the setting of specific compiler options or pragmas. Unless noted otherwise, all feature-related macros are protected (the compiler will issue a warning if you try to undefine or redefine them).

Feature-related macros are discussed in the following sections:

- “Macros related to compiler option settings”
- “Macros related to architecture settings” on page 330
- “Macros related to language levels” on page 331

Macros related to compiler option settings

The following macros can be tested for various features, including source input characteristics, output file characteristics, optimization, and so on. All of these macros are predefined by a specific compiler option or suboption, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined.

Table 37. General option-related predefined macros






Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
<code>__ALTIVEC__</code>	Indicates support for vector data types. (unprotected)	1	<code>-qaltivec</code>
<code>__64BIT__</code>	Indicates that 64-bit compilation mode is in effect.	1	<code>-q64</code>
<code>_CHAR_SIGNED</code> , <code>__CHAR_SIGNED__</code>	Indicates that the default character type is signed char.	1	<code>-qchars=signed</code>
<code>_CHAR_UNSIGNED</code> , <code>__CHAR_UNSIGNED__</code>	Indicates that the default character type is unsigned char.	1	<code>-qchars=unsigned</code>
 <code>__EXCEPTIONS</code>	Indicates that C++ exception handling is enabled.	1	<code>-qeh</code>

Table 37. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
__IBM_GCC_ASM	Indicates support for GCC inline asm statements.	1	<div> <div>C</div> -qasm=gcc and -qlanglvl=extc99 extc89 extended or -qkeyword=asm </div> <div> <div>C++</div> -qasm=gcc and -qlanglvl=extended </div>
		0	<div> <div>C</div> -qnoasm and -qlanglvl=extc99 extc89 extended or -qkeyword=asm </div> <div> <div>C++</div> -qnoasm and -qlanglvl=extended </div>
<div>C++</div> __IBM_STDCPP_ASM	Indicates that support for GCC inline asm statements is disabled.	0	-qnoasm=stdcpp
__IBM_UTF_LITERAL	Indicates support for UTF-16 and UTF-32 string literals.	1	-qutf
<div>C++</div> __IGNERRNO__	Indicates that system calls do not modify errno, thereby enabling certain compiler optimizations.	1	-qignerrno
<div>C++</div> __INITAUTO__	Indicates the value to which automatic variables which are not explicitly initialized in the source program are to be initialized.	The two-digit hexadecimal value specified in the -qinitauto compiler option.	-qinitauto=hex value
<div>C++</div> __INITAUTO_W__	Indicates the value to which automatic variables which are not explicitly initialized in the source program are to be initialized.	An eight-digit hexadecimal corresponding to the value specified in the -qinitauto compiler option repeated 4 times.	-qinitauto=hex value
<div>C++</div> __LIBANSI__	Indicates that calls to functions whose names match those in the C Standard Library are in fact the C library functions, enabling certain compiler optimizations.	1	-qlibansi

Table 37. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
__LONGDOUBLE64	Indicates that the size of a long double type is 64 bits.	1	Always predefined for the SPU.
__LONGDOUBLE128, __LONG_DOUBLE_128__	Indicates that the size of a long double type is 128 bits.	1	Always predefined for the PPU.
__OPTIMIZE__	Indicates the level of optimization in effect.	2	-O -O2
		3	-O3 -O4 -O5
__OPTIMIZE_SIZE__	Indicates that optimization for code size is in effect.	1	-O -O2 -O3 -O4 -O5 and -qcompact
 __RTTI_DYNAMIC_CAST__	Indicates that runtime type identification information for the dynamic_cast operator is generated.	1	-qrtti
 __RTTI_TYPE_INFO__	Indicates that runtime type identification information for the typeid operator is generated.	1	-qrtti
 __NO_RTTI__	Indicates that runtime type identification information is disabled.	1	-qnortti
 __TEMPINC__	Indicates that the compiler is using the template-implementation file method of resolving template functions.	1	-qtempinc
__VEC__	Indicates support for vector data types.	10205	-qaltivec

Macros related to architecture settings

The following macros can be tested for target architecture settings. All of these macros are predefined to a value of 1 by a **-qarch** compiler option setting, or any other compiler option that implies that setting. If the **-qarch** suboption enabling the feature is not in effect, then the macro is undefined.

Table 38. **-qarch**-related macros

Macro name	Description	Predefined by the following -qarch suboptions
__ARCH_CELLPPU	Indicates that the application is targeted to run on the PPU.	cellppu
__ARCH_CELLSPU	Indicates that the application is targeted to run on the SPU.	cellspu
__ARCH_CELLSPU_EDP	Indicates that the application is targeted to run on the enhanced CBEA-compliant processor with a double precision SPU.	edp
__ARCH_COM	Indicates that the application is targeted to run on any Power processor.	cellppu

Table 38. **-qarch**-related macros (continued)

Macro name	Description	Predefined by the following -qarch suboptions
<code>_ARCH_PPC</code>	Indicates that the application is targeted to run on any Power processor.	cellppu
<code>_ARCH_PPC64</code>	Indicates that the application is targeted to run on Power processors with 64-bit support.	cellppu
<code>_ARCH_PPCGR</code>	Indicates that the application is targeted to run on Power processors with graphics support.	cellppu
<code>_ARCH_PPC64GR</code>	Indicates that the application is targeted to run on Power processors with 64-bit and graphics support.	cellppu
<code>_ARCH_PPC64GRSQ</code>	Indicates that the application is targeted to run on Power processors with 64-bit, graphics, and square root support.	cellppu
<code>_ARCH_PPC64V</code>	Indicates that the application is targeted to run on Power processors with 64-bit and vector processing support.	cellppu
<code>__PPU__</code>	Indicates that the application is targeted to run on the PPU.	cellppu
<code>__SPU__</code>	Indicates that the application is targeted to run on the SPU.	cellspu
<code>__SPU_EDP__</code>	Indicates that the application is targeted to run on the enhanced CBEA-compliant processor with a double precision SPU.	edp

Macros related to language levels

The following macros can be tested for C99 features, features related to GNU C or C++, and other IBM language extensions. All of these macros are predefined to a value of 1 by a specific language level, represented by a suboption of the **-qlanglvl** compiler option, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined. For descriptions of the features related to these macros, see the *XL C/C++ Language Reference*.

Table 39. Predefined macros for language features






Predefined macro name	Description	Predefined when the following language level is in effect
 <code>__BOOL__</code>	Indicates that the <code>bool</code> keyword is accepted.	Always defined except when -qnokeyword=bool is in effect.
 <code>__C99_BOOL</code>	Indicates support for the <code>_Bool</code> data type.	stdc99 extc99 extc89 extended
 <code>__C99_COMPLEX</code>	Indicates support for complex data types.	stdc99 extc99 extc89 extended
 <code>__C99_CPLUSCMT</code>	Indicates support for C++ style comments	stdc99 extc99 (also -qcpluspluscmt)
 <code>__C99_COMPOUND_LITERAL</code>	Indicates support for compound literals.	stdc99 extc99 extc89 extended

Table 39. Predefined macros for language features (continued)





















Predefined macro name	Description	Predefined when the following language level is in effect
 <code>__C99_DESIGNATED_INITIALIZER</code>	Indicates support for designated initialization.	stdc99 extc99 extc89 extended
 <code>__C99_DUP_TYPE_QUALIFIER</code>	Indicates support for duplicated type qualifiers.	stdc99 extc99 extc89 extended
<code>__C99_EMPTY_MACRO_ARGUMENTS</code>	Indicates support for empty macro arguments.	 stdc99 extc99 extc89 extended  extended
 <code>__C99_FLEXIBLE_ARRAY_MEMBER</code>	Indicates support for flexible array members.	stdc99 extc99 extc89 extended
<code>__C99_FUNC__</code>	Indicates support for the <code>__func__</code> predefined identifier.	 stdc99 extc99 extc89 extended  extended <code>c99_func__</code>
<code>__C99_HEX_FLOAT_CONST</code>	Indicates support for hexadecimal floating constants.	 stdc99 extc99 extc89 extended  extended <code>c99hexfloat</code>
 <code>__C99_INLINE</code>	Indicates support for the <code>inline</code> function specifier.	stdc99 extc99 (also -qkeyword=inline)
 <code>__C99_LLONG</code>	Indicates support for C99-style long long data types.	stdc99 extc99
<code>__C99_MACRO_WITH_VA_ARGS</code>	Indicates support for function-like macros with variable arguments.	 stdc99 extc99 extc89 extended  extended <code>varargmacros</code> extended0x
<code>__C99_MAX_LINE_NUMBER</code>	Indicates that the maximum line number is 2147483647.	 stdc99 extc99 extc89 extended  extended0x
 <code>__C99_MIXED_DECL_AND_CODE</code>	Indicates support for mixed declaration and code.	stdc99 extc99 extc89 extended
<code>__C99_MIXED_STRING_CONCAT</code>	Indicates support for concatenation of wide string and non-wide string literals.	 stdc99 extc99 extc89 extended  extended0x
 <code>__C99_NON_LVALUE_ARRAY_SUB</code>	Indicates support for non-lvalue subscripts for arrays.	stdc99 extc99 extc89 extended
 <code>__C99_NON_CONST_AGGG_INITIALIZERS</code>	Indicates support for non-constant aggregate initializers.	stdc99 extc99 extc89 extended











Table 39. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
<code>__C99_PRAGMA_OPERATOR</code>	Indicates support for the <code>_Pragma</code> operator.	<div>► C</div> <div>stdc99 extc99 extc89 extended</div> <div>► C++</div> <div>extended extended0x</div>
► C <code>__C99_REQUIRE_FUNC_DECL</code>	Indicates that implicit function declaration is not supported.	stdc99
<code>__C99_RESTRICT</code>	Indicates support for the C99 <code>restrict</code> qualifier.	<div>► C</div> <div>stdc99 extc99 (also -qkeyword=restrict)</div> <div>► C++</div> <div>extended (also -qkeyword=restrict)</div>
► C <code>__C99_STATIC_ARRAY_SIZE</code>	Indicates support for the <code>static</code> keyword in array parameters to functions.	stdc99 extc99 extc89 extended
► C <code>__C99_STD_PRAGMAS</code>	Indicates support for standard pragmas.	stdc99 extc99 extc89 extended
► C <code>__C99_TGMATH</code>	Indicates support for type-generic macros in <code>tgmath.h</code> .	stdc99 extc99 extc89 extended
<code>__C99_UCN</code>	Indicates support for universal character names.	<div>► C</div> <div>stdc99 extc99 ucs</div> <div>► C++</div> <div>ucs</div>
► C <code>__C99_VAR_LEN_ARRAY</code>	Indicates support for variable length arrays.	stdc99 extc99 extc89 extended
► C++ <code>__C99_VARIABLE_LENGTH_ARRAY</code>	Indicates support for variable length arrays.	extended c99vla
<code>__DIGRAPHS__</code>	Indicates support for digraphs.	<div>► C</div> <div>stdc99 extc99 extc89 extended (also -qdigraph)</div> <div>► C++</div> <div>extended (also -qdigraph)</div>
► C <code>__EXTENDED__</code>	Indicates that language extensions are supported.	extended
<code>__IBM__ALIGN</code>	Indicates support for the <code>__align</code> specifier.	► C++ Always defined except when -qnokeyword=__alignof is specified
<code>__IBM_ALIGNOF__</code>	Indicates support for the <code>__alignof__</code> operator.	<div>► C</div> <div>extc99 extc89 extended</div> <div>► C++</div> <div>extended</div>

Table 39. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
<code>__IBM_ATTRIBUTES</code>	Indicates support for type, variable, and function attributes.	<div> <div>C</div> <div>extc99 extc89 extended</div> </div> <div> <div>C++</div> <div>extended</div> </div>
<code>__IBM_COMPUTED_GOTO</code>	Indicates support for computed goto statements.	<div> <div>C</div> <div>extc99 extc89 extended</div> </div> <div> <div>C++</div> <div>extended gnu_computedgoto</div> </div>
<code>__IBM_EXTENSION_KEYWORD</code>	Indicates support for the <code>__extension__</code> keyword.	<div> <div>C</div> <div>extc99 extc89 extended</div> </div> <div> <div>C++</div> <div>extended</div> </div>
<code>__IBM_GCC_INLINE__</code>	Indicates support for the GCC <code>__inline__</code> specifier.	<div> <div>C</div> <div>extc99 extc89 extended</div> </div> <div> <div>C++</div> <div>extended</div> </div>
<div>C</div> <code>__IBM_DOLLAR_IN_ID</code>	Indicates support for dollar signs in identifiers.	extc99 extc89 extended
<div>C</div> <code>__IBM_GENERALIZED_LVALUE</code>	Indicates support for generalized lvalues.	extc99 extc89 extended
<code>__IBM_INCLUDE_NEXT</code>	Indicates support for the <code>#include_next</code> preprocessing directive.	<div> <div>C</div> <div>Always defined</div> </div> <div> <div>C++</div> <div>Always defined except when <code>-qlanglvl=nognu_include_next</code> is in effect.</div> </div>
<code>__IBM_LABEL_VALUE</code>	Indicates support for labels as values.	<div> <div>C</div> <div>extc99 extc89 extended</div> </div> <div> <div>C++</div> <div>extended gnu_labelvalue</div> </div>
<code>__IBM_LOCAL_LABEL</code>	Indicates support for local labels.	<div> <div>C</div> <div>extc99 extc89 extended</div> </div> <div> <div>C++</div> <div>extended gnu_locallabel</div> </div>
<code>__IBM_MACRO_WITH_VA_ARGS</code>	Indicates support for variadic macro extensions.	<div> <div>C</div> <div>extc99 extc89 extended</div> </div> <div> <div>C++</div> <div>extended gnu_varargmacros</div> </div>
<div>C</div> <code>__IBM_NESTED_FUNCTION</code>	Indicates support for nested functions.	extc99 extc89 extended

Table 39. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
 <code>__IBM_PP_PREDICATE</code>	Indicates support for <code>#assert</code> , <code>#unassert</code> , <code>#cpu</code> , <code>#machine</code> , and <code>#system</code> preprocessing directives.	extc99 extc89 extended
 <code>__IBM_PP_WARNING</code>	Indicates support for the <code>#warning</code> preprocessing directive.	extc99 extc89 extended
 <code>__IBM_REGISTER_VARS</code>	Indicates support for variables in specified registers.	Always defined.
 <code>__IBM_REGISTER_VARIABLES</code>	Indicates support for variables in specified registers.	Always defined.
<code>__IBM__TYPEOF__</code>	Indicates support for the <code>__typeof__</code> or <code>typeof</code> keyword.	 always defined  extended (Also <code>-qkeyword=typeof</code>)
<code>__LONG_LONG</code>	Indicates support for IBM long long data types.	 extended extc89 (also <code>-qlonglong</code>)  extended (also <code>-qlonglong</code>)
<code>__STDC__</code>	Indicates that the compiler conforms to the ANSI/ISO C standard.	Predefined to 1 if ANSI/ISO C standard conformance is in effect.
 <code>__STDC_HOSTED__</code>	Indicates that the implementation is a hosted implementation of the ANSI/ISO C standard. (That is, the hosted environment has all the facilities of the standard C available).	Predefined to 1 if this is the case; otherwise, defined to 0.
 <code>__STDC_VERSION__</code>	Indicates the version of ANSI/ISO C standard which the compiler conforms to.	The format is <code>yyyymmL</code> . (For example, the format is <code>199901L</code> for C99.)

Examples of predefined macros

This example illustrates use of the `__FUNCTION__` and the `__C99_FUNC__` macros to test for the availability of the C99 `__func__` identifier to return the current function name:

```
#include <stdio.h>
```

```
#if defined(__C99_FUNC__)
#define PRINT_FUNC_NAME() printf (" In function %s \n", __func__);
#elif defined(__FUNCTION__)
#define PRINT_FUNC_NAME() printf (" In function %s \n", __FUNCTION__);
#else
#define PRINT_FUNC_NAME() printf (" Function name unavailable\n");
#endif
```

```
void foo(void);
```

```

int main(int argc, char **argv)
{
    int k = 1;
    PRINT_FUNC_NAME();
    foo();
    return 0;
}

void foo (void)
{
    PRINT_FUNC_NAME();
    return;
}


```

The output of this example is:

```

In function main
In function foo

```

 This example illustrates use of the `__FUNCTION__` macro in a C++ program with virtual functions.

```

#include <stdio.h>
class X { public: virtual void func() = 0;};

class Y : public X {
    public: void func() { printf("In function %s \n", __FUNCTION__);}
};

int main() {
    Y aaa;
    aaa.func();
}

```

The output of this example is:

```


In function Y::func()

```

Chapter 6. Macros independent of compiler options

The following macros are undefined by default and are not related to specific compiler options.


Table 40. Macros independent of compiler options

Macro name	Description
 <code>__IBM_ALLOW_OVERRIDE_PLACEMENT_NEW</code>	Can be used to obtain the behavior of global placement new operators. By default, this macro is not predefined, and the Standard C++ Library provides the inlined version of global placement new operators, which can assist with improving performance.

Chapter 7. Compiler built-in functions

A built-in function is a coding extension to C and C++ that allows a programmer to use the syntax of C function calls and C variables to access the instruction set of the processor of the compiling machine. IBM Power architectures have special instructions that enable the development of highly optimized applications. Access to some Power instructions cannot be generated using the standard constructs of the C and C++ languages. Other instructions can be generated through standard constructs, but using built-in functions allows exact control of the generated code. Inline assembly language programming, which uses these instructions directly, is not fully supported by XL C/C++ and other compilers. Furthermore, the technique can be time-consuming to implement.

As an alternative to managing hardware registers through assembly language, XL C/C++ built-in functions provide access to the optimized Power instruction set and allow the compiler to optimize the instruction scheduling.

 To call any of the XL C/C++ built-in functions in C++, you must include the header file `builtins.h` in your source code.

For code compiled to target the PPU, the compiler supports all vector processing functions defined by the AltiVec specification. For detailed descriptions of all of these built-in functions, see the *AltiVec Technology Programming Interface Manual*, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.

The compiler additionally supports built-in functions specific to the Cell Broadband Engine Architecture Synergistic Processor Unit (SPU). To call any of these functions, you must include the header file `spu_intrinsics.h` in your source. For detailed descriptions of the SPU built-in functions, see the document *IBM C/C++ Language Extensions for Cell Broadband Engine Architecture*, available at <http://publib.boulder.ibm.com/infocenter/systems/topic/eiccg/eiccgkickoff.htm>.

The following sections describe the available built-in functions.

- “Fixed-point built-in functions”
- “Floating-point built-in functions” on page 345
- “Synchronization and atomic built-in functions (PPU only)” on page 352
- “Cache-related built-in functions (PPU only)” on page 360
- “Block-related built-in functions” on page 361
- “Miscellaneous built-in functions” on page 362

Fixed-point built-in functions

Fixed-point built-in functions are grouped into the following categories:

- “Absolute value functions” on page 340
- “Assert functions” on page 340
- “Count zero functions” on page 340
- “Load functions” on page 341
- “Multiply functions” on page 341
- “Population count functions” on page 342
- “Rotate functions” on page 342

- “Store functions” on page 344
- “Trap functions” on page 344

Absolute value functions

__labs, __llabs

Purpose

Absolute Value Long, Absolute Value Long Long

Returns the absolute value of the argument.

Prototype

```
signed long __labs (signed long);
```

```
signed long long __llabs (signed long long);
```

Assert functions

__assert1, __assert2 (PPU only)

Purpose

Generates trap instructions.

Prototype

```
int __assert1 (int, int, int);
```

```
void __assert2 (int);
```

Count zero functions

__cntlz4, __cntlz8

Purpose

Count Leading Zeros, 4/8-byte integer

Prototype

```
unsigned int __cntlz4 (unsigned int);
```

```
unsigned int __cntlz8 (unsigned long long);
```

__cnttz4, __cnttz8

Purpose

Count Trailing Zeros, 4/8-byte integer

Prototype

```
unsigned int __cnttz4 (unsigned int);
```

```
unsigned int __cnttz8 (unsigned long long);
```

Load functions

__load2r, __load4r, __load8r (PPU only)

Purpose

Load Halfword Byte Reversed, Load Word Byte Reversed, Load Doubleword Byte Reversed

Prototype

```
unsigned short __load2r (unsigned short*);
```

```
unsigned int __load4r (unsigned int*);
```

```
unsigned long long __load8r (unsigned long long*);
```

Multiply functions

__mulhd, __mulhdu (PPU only)

Purpose

Multiply High Doubleword Signed, Multiply High Doubleword Unsigned

Returns the highorder 64 bits of the 128bit product of the two parameters.

Prototype

```
long long int __mulhd ( long int, long int);
```

```
unsigned long long int __mulhdu (unsigned long int, unsigned long int);
```

Usage

Valid only in 64-bit mode.

__mulhw, __mulhwu (PPU only)

Purpose

Multiply High Word Signed, Multiply High Word Unsigned

Returns the highorder 32 bits of the 64bit product of the two parameters.

Prototype

```
int __mulhw (int, int);
```

```
unsigned int __mulhwu (unsigned int, unsigned int);
```

Population count functions

__popcnt4, __popcnt8

Purpose

Population Count, 4/8-byte integer

Returns the number of bits set for a 32/64-bit integer.

Prototype

```
int __popcnt4 (unsigned int);
```

```
int __popcnt8 (unsigned long long);
```

__popcntb

Purpose

Population Count Byte

Counts the 1 bits in each byte of the parameter and places that count into the corresponding byte of the result.

Prototype

```
unsigned long __popcntb(unsigned long);
```

__poppar4, __poppar8

Purpose

Population Parity, 4/8-byte integer

Checks whether the number of bits set in a 32/64-bit integer is an even or odd number.

Prototype

```
int __poppar4(unsigned int);
```

```
int __poppar8(unsigned long long);
```

Return value

Returns 1 if the number of bits set in the input parameter is odd. Returns 0 otherwise.

Rotate functions

__rdlam (PPU only)

Purpose

Rotate Double Left and AND with Mask

Rotates the contents of *rs* left *shift* bits, and ANDs the rotated data with the *mask*.

Prototype

```
unsigned long long __rdlam (unsigned long long rs, unsigned int shift,  
unsigned long long mask);
```

Parameters

mask

Must be a constant that represents a contiguous bit field.

__rldimi, __rlwimi (PPU only)

Purpose

Rotate Left Doubleword Immediate then Mask Insert, Rotate Left Word Immediate then Mask Insert

Rotates *rs* left *shift* bits then inserts *rs* into *is* under bit mask *mask*.

Prototype

```
unsigned long long __rldimi (unsigned long long rs, unsigned long long is,  
unsigned int shift, unsigned long long mask);
```

```
unsigned int __rlwimi (unsigned int rs, unsigned int is, unsigned int shift,  
unsigned int mask);
```

Parameters

shift

A constant value 0 to 63 (__rldimi) or 31 (__rlwimi).

mask

Must be a constant that represents a contiguous bit field.

__rlwnm (PPU only)

Purpose

Rotate Left Word then AND with Mask

Rotates *rs* left *shift* bits, then ANDs *rs* with bit mask *mask*.

Prototype

```
unsigned int __rlwnm (unsigned int rs, unsigned int shift, unsigned int mask);
```

Parameters

mask

Must be a constant that represents a contiguous bit field.

__rotatel4, __rotatel8 (PPU only)

Purpose

Rotate Left Word, Rotate Left Doubleword

Rotates *rs* left *shift* bits.

Prototype

```
unsigned int __rotatel4 (unsigned int rs, unsigned int shift);  
  
unsigned long long __rotatel8 (unsigned long long rs, unsigned long long  
shift);
```

Store functions

__store2r, __store4r, __store8r (PPU only)

Purpose

Store 2/4/8-byte Register

Prototype

```
void __store2r (unsigned short, unsigned short*);  
  
void __store4r (unsigned int, unsigned int*);  
  
void __store8r (unsigned long long, unsigned long long*);
```

Trap functions

__tdw, __tw (PPU only)

Purpose

Trap Doubleword, Trap Word

Compares parameter *a* with parameter *b*. This comparison results in five conditions which are ANDed with a 5-bit constant *TO*. If the result is not 0 the system trap handler is invoked.

Prototype

```
void __tdw ( long a, long b, unsigned int TO);  
  
void __tw (int a, int b, unsigned int TO);
```

Parameters

TO

A value of 0 to 31 inclusive. Each bit position, if set, indicates one or more of the following possible conditions:

0 (high-order bit)

a is less than *b*, using signed comparison.

1 *a* is greater than *b*, using signed comparison.

2 *a* is equal to *b*

3 *a* is less than *b*, using unsigned comparison.

4 (low-order bit)

a is greater than *b*, using unsigned comparison.

Usage

`__tdw` is valid only in 64-bit mode.

`__trap`, `__trapd` (PPU only)

Purpose

Trap if the Parameter is not Zero, Trap if the Parameter is not Zero Doubleword

Prototype

```
void __trap (int);  
  
void __trapd ( long);
```

Usage

`__trapd` is valid only in 64-bit mode.

Floating-point built-in functions

Floating-point built-in functions are grouped into the following categories:

- “Absolute value functions” on page 340
- “Conversion functions” on page 346
- “FPSCR functions” on page 347
- “Multiply-add/subtract functions” on page 349
- “Reciprocal estimate functions” on page 350
- “Select functions” on page 350
- “Square root functions” on page 351
- “Software division functions” on page 351

Absolute value functions

`__fabss`

Purpose

Floating Absolute Value Single

Returns the absolute value of the argument.

Prototype

```
float __fabss (float);
```

`__fnabs` (PPU only)

Purpose

Floating Negative Absolute Value, Floating Negative Absolute Value Single

Returns the negative absolute value of the argument.

Prototype

```
double __fnabs (double);
```

```
float __fnabss (float);
```

Conversion functions

__cplx, __cplx, __cplx

Purpose

Converts two real parameters into a single complex value.

Prototype

```
double _Complex __cplx (double, double);
```

```
float _Complex __cplx (float, float);
```

```
long double _Complex __cplx (long double, long double);
```

__fcid (PPU only)

Purpose

Floating Convert from Integer Doubleword

Converts a 64-bit signed integer stored in a double to a double-precision floating-point value.

Prototype

```
double __fcid (double);
```

__fctid (PPU only)

Purpose

Floating Convert to Integer Doubleword

Converts a double-precision argument to a 64-bit signed integer, using the current rounding mode, and returns the result in a double.

Prototype

```
double __fctid (double);
```

__fctidz (PPU only)

Purpose

Floating Convert to Integer Doubleword with Rounding towards Zero

Converts a double-precision argument to a 64-bit signed integer, using the rounding mode round-toward-zero, and returns the result in a double.

Prototype

```
double __fctidz (double);
```

__fctiw (PPU only)

Purpose

Floating Convert to Integer Word

Converts a double-precision argument to a 32-bit signed integer, using the current rounding mode, and returns the result in a double.

Prototype

```
double __fctiw (double);
```

__fctiwz (PPU only)

Purpose

Floating Convert to Integer Word with Rounding towards Zero

Converts a double-precision argument to a 32-bit signed integer, using the rounding mode round-toward-zero, and returns the result in a double.

Prototype

```
double __fctiwz (double);
```

__ibm2gccldbl, __ibm2gccldbl_cmplx (PPU only)

Purpose

Converts IBM-style long double data types to GCC long doubles.

Prototype

```
long double __ibm2gccldbl (long double);
```

```
_Complex long double __ibm2gccldbl_cmplx (_Complex long double);
```

Return value

The translated result conforms to GCC requirements for long doubles. However, long double computations performed in IBM-compiled code may not produce bitwise identical results to those obtained purely by GCC.

FPSCR functions

__mtfsb0 (PPU only)

Purpose

Move to Floating Point Status/Control Register (FPSCR) Bit 0

Sets bit *bt* of the FPSCR to 0.

Prototype

```
void __mtfsb0 (unsigned int bt);
```

Parameters

bt Must be a constant with a value of 0 to 31.

__mtfsb1 (PPU only)

Purpose

Move to FPSCR Bit 1

Sets bit *bt* of the FPSCR to 1.

Prototype

```
void __mtfsb1 (unsigned int bt);
```

Parameters

bt Must be a constant with a value of 0 to 31.

__mtfsf (PPU only)

Purpose

Move to FPSCR Fields

Places the contents of *frb* into the FPSCR under control of the field mask specified by *flm*. The field mask *flm* identifies the 4bit fields of the FPSCR affected.

Prototype

```
void __mtfsf (unsigned int flm, unsigned int frb);
```

Parameters

flm

Must be a constant 8-bit mask.

__mtfsfi (PPU only)

Purpose

Move to FPSCR Field Immediate

Places the value of *u* into the FPSCR field specified by *bf*.

Prototype

```
void __mtfsfi (unsigned int bf, unsigned int u);
```

Parameters

bf Must be a constant with a value of 0 to 7.

u Must be a constant with a value of 0 to 15.

__readflm (PPU only)

Purpose

Returns a 64-bit double precision floating point, whose 32 low order bits contain the contents of the FPSCR. The 32 low order bits are bits 32 - 63 counting from the highest order bit.

Prototype

```
double __readflm (void);
```

__setflm (PPU only)

Purpose

Takes a double precision floating point number and places the lower 32 bits in the FPSCR. The 32 low order bits are bits 32 - 63 counting from the highest order bit. Returns the previous contents of the FPSCR.

Prototype

```
double __setflm (double);
```

__setrnd (PPU only)

Purpose

Sets the rounding mode.

Prototype

```
double __setrnd (int mode);
```

Parameters

The allowable values for *mode* are:

- 0 — round to nearest
- 1 — round to zero
- 2 — round to +infinity
- 3 — round to -infinity

Multiply-add/subtract functions

__fmadd, __fmadds (PPU only)

Purpose

Floating Multiply-Add, Floating Multiply-Add Single

Multiplies the first two arguments, adds the third argument, and returns the result.

Prototype

```
double __fmadd (double, double, double);
```

```
float __fmadds (float, float, float);
```

__fmsub, __fmsubs (PPU only)

Purpose

Floating Multiply-Subtract, Floating Multiply-Subtract Single

Multiplies the first two arguments, subtracts the third argument and returns the result.

Prototype

```
double __fmsub (double, double, double);
```

```
float __fmsubs (float, float, float);
```

__fnmadd, __fnmadds (PPU only)

Purpose

Floating Negative Multiply-Add, Floating Negative Multiply-Add Single

Multiplies the first two arguments, adds the third argument, and negates the result.

Prototype

```
double __fnmadd (double, double, double);
```

```
float __fnmadds (float, float, float);
```

__fnmsub, __fnmsubs (PPU only)

Purpose

Floating Negative Multiply-Subtract

Multiplies the first two arguments, subtracts the third argument, and negates the result.

Prototype

```
double __fnmsub (double, double, double);
```

```
float __fnmsubs (float, float, float);
```

Reciprocal estimate functions

See also “Square root functions” on page 351.

__fres (PPU only)

Purpose

Floating Reciprocal Estimate Single

Prototype

```
float __fres (float);
```

Select functions

__fsel, __fsels (PPU only)

Purpose

Floating Select, Floating Select Single

Returns the second argument if the first argument is greater than or equal to zero; returns the third argument otherwise.

Prototype

```
double __fsel (double, double, double);
```

```
float __fsels (float, float, float);
```

Square root functions

__frsqrt (PPU only)

Purpose

Floating Reciprocal Square Root Estimate

Prototype

```
double __frsqrt (double);
```

__fsqrt, __fsqrts

Purpose

Floating Square Root, Floating Square Root Single

Prototype

```
double __fsqrt (double);
```

```
float __fsqrts (float);
```

Software division functions

__swdiv, __swdivs

Purpose

Software Divide, Software Divide Single

Divides the first argument by the second argument and returns the result.

Prototype

```
double __swdiv (double, double);
```

```
float __swdivs (float, float);
```

__swdiv_nochk, __swdivs_nochk

Purpose

Software Divide No Check, Software Divide No Check Single

Divides the first argument by the second argument, without performing range checking, and returns the result.

Prototype

```
double __swdiv_nochk (double a, double b);
```

```
float __swdivs_nochk (float a, float b);
```

Parameters

- a* Must not equal infinity. When **-qstrict** is in effect, *a* must have an absolute value greater than 2^{-970} and less than infinity.
- b* Must not equal infinity, zero, or denormalized values. When **-qstrict** is in effect, *b* must have an absolute value greater than 2^{-1022} and less than 2^{1021} .

Return value

The result must not be equal to positive or negative infinity. When **-qstrict** in effect, the result must have an absolute value greater than 2^{-1021} and less than 2^{1023} .

Usage

This function can provide better performance than the normal divide operator or the `__sdiv` built-in function in situations where division is performed repeatedly in a loop and when arguments are within the permitted ranges.

Store functions

`__stfiw` (PPU only)

Purpose

Store Floating Point as Integer Word

Stores the contents of the loworder 32 bits of *value*, without conversion, into the word in storage addressed by *addr*.

Prototype

```
void __stfiw (const int* addr, double value);
```

Synchronization and atomic built-in functions (PPU only)

Synchronization and atomic built-in functions are grouped into the following categories:

- “Check lock functions”
- “Clear lock functions” on page 354
- “Compare and swap functions” on page 355
- “Fetch functions” on page 356
- “Load functions” on page 357
- “Store functions” on page 358
- “Synchronization functions” on page 359

Check lock functions

`__check_lock_mp`, `__check_lockd_mp`

Purpose

Check Lock on Multiprocessor Systems, Check Lock Doubleword on Multiprocessor Systems

Conditionally updates a single word or doubleword variable atomically.

Prototype

```
unsigned int __check_lock_mp (const int* addr, int old_value, int new_value);
```

```
unsigned int __check_lockd_mp (const long* addr, long old_value, long  
new_value);
```

Parameters

addr

The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word or on an 8-byte boundary for a doubleword.

old_value

The old value to be checked against the current value in *addr*.

new_value

The new value to be conditionally assigned to the variable in *addr*,

Return value

Returns false (0) if the value in *addr* was equal to *old_value* and has been set to the *new_value*. Returns true (1) if the value in *addr* was not equal to *old_value* and has been left unchanged.

Usage

`__check_lockd_mp` is valid only in 64-bit mode.

`__check_lock_up`, `__check_lockd_up` Purpose

Check Lock on Uniprocessor Systems, Check Lock Doubleword on Uniprocessor Systems

Conditionally updates a single word or doubleword variable atomically.

Prototype

```
unsigned int __check_lock_up (const int* addr, int old_value, int new_value);
```

```
unsigned int __check_lockd_up (const long* addr, long old_value, long  
new_value);
```

Parameters

addr

The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

old_value

The old value to be checked against the current value in *addr*.

new_value

The new value to be conditionally assigned to the variable in *addr*,

Return value

Returns false (0) if the value in *addr* was equal to *old_value* and has been set to the new value. Returns true (1) if the value in *addr* was not equal to *old_value* and has been left unchanged.

Usage

`__check_lockd_up` is valid only in 64-bit mode.

Clear lock functions

`__clear_lock_mp`, `__clear_lockd_mp`

Purpose

Clear Lock on Multiprocessor Systems, Clear Lock Doubleword on Multiprocessor Systems

Atomic store of the *value* into the variable at the address *addr*.

Prototype

```
void __clear_lock_mp (const int* addr, int value);
```

```
void __clear_lockd_mp (const long* addr, long value);
```

Parameters

addr

The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value

The new value to be assigned to the variable in *addr*,

Usage

`__clear_lockd_mp` is only valid in 64-bit mode.

`__clear_lock_up`, `__clear_lockd_up`

Purpose

Clear Lock on Uniprocessor Systems, Clear Lock Doubleword on Uniprocessor Systems

Atomic store of the *value* into the variable at the address *addr*.

Prototype

```
void __clear_lock_up (const int* addr, int value);
```

```
void __clear_lockd_up (const long* addr, long value);
```

Parameters

addr

The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value

The new value to be assigned to the variable in *addr*.

Usage

`__clear_lockd_up` is only valid in 64-bit mode.

Compare and swap functions

`__compare_and_swap`, `__compare_and_swaplp`

Purpose

Conditionally updates a single word or doubleword variable atomically.

Prototype

```
int __compare_and_swap (volatile int* addr, int* old_val_addr, int new_val);
```

```
int __compare_and_swaplp (volatile long* addr, long* old_val_addr, long  
new_val);
```

Parameters

addr

The address of the variable to be copied. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

old_val_addr

The memory location into which the value in *addr* is to be copied.

new_val

The value to be conditionally assigned to the variable in *addr*,

Return value

Returns true (1) if the value in *addr* was equal to *old_value* and has been set to the new value. Returns false (0) if the value in *addr* was not equal to *old_value* and has been left unchanged. In either case, the contents of the memory location specified by *addr* are copied into the memory location specified by *old_val_addr*.

Usage

The `__compare_and_swap` function is useful when a single word value must be updated only if it has not been changed since it was last read. If you use `__compare_and_swap` as a locking primitive, insert a call to the `__isync` built-in function at the start of any critical sections.

`__compare_and_swaplp` is valid only in 64-bit mode.

Fetch functions

__fetch_and_and, __fetch_and_andlp

Purpose

Clears bits in the word or doubleword specified by *addr* by AND-ing that value with the value specified by *val*, in a single atomic operation, and returns the original value of *addr*.

Prototype

```
unsigned int __fetch_and_and (volatile unsigned int* addr, unsigned int val);
```

```
unsigned long __fetch_and_andlp (volatile unsigned long* addr, unsigned long val);
```

Parameters

addr

The address of the variable to be ANDed. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value

The value by which the value in *addr* is to be ANDed.

Usage

This operation is useful when a variable containing bit flags is shared between several threads or processes.

`__fetch_and_andlp` is valid only in 64-bit mode.

__fetch_and_or, __fetch_and_orlp

Purpose

Sets bits in the word or doubleword specified by *addr* by OR-ing that value with the value specified *val*, in a single atomic operation, and returns the original value of *addr*.

Prototype

```
unsigned int __fetch_and_or (volatile unsigned int* addr, unsigned int val);
```

```
unsigned long __fetch_and_orlp (volatile unsigned long* addr, unsigned long val);
```

Parameters

addr

The address of the variable to be ORed. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value

The value by which the value in *addr* is to be ORed.

Usage

This operation is useful when a variable containing bit flags is shared between several threads or processes.

`__fetch_and_orlp` is valid only in 64-bit mode.

`__fetch_and_swap`, `__fetch_and_swaplp` Purpose

Sets the word or doubleword specified by *addr* to the value of *val* and returns the original value of *addr*, in a single atomic operation.

Prototype

```
unsigned int __fetch_and_swap (volatile unsigned int* addr, unsigned int val);
```

```
unsigned long __fetch_and_swaplp (volatile unsigned long* addr, unsigned  
long val);
```

Parameters

addr

The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value

The value which is to be assigned to *addr*.

Usage

This operation is useful when a variable is shared between several threads or processes, and one thread needs to update the value of the variable without losing the value that was originally stored in the location.

`__fetch_and_swaplp` is valid only in 64-bit mode.

Load functions

`__ldarx`, `__lwarx` Purpose

Load Doubleword and Reserve Indexed, Load Word and Reserve Indexed

Loads the value from the memory location specified by *addr* and returns the result. For `__lwarx`, in 64-bit mode, the compiler returns the sign-extended result.

Prototype

```
long __ldarx (volatile long* addr);
```

```
int __lwarx (volatile int* addr);
```

Parameters

addr

The address of the value to be loaded. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

Usage

This function can be used with a subsequent `__stdcx` (or `__stwcx`) built-in function to implement a read-modify-write on a specified memory location. The two built-in functions work together to ensure that if the store is successfully performed, no other processor or mechanism can modify the target doubleword between the time the `__ldarx` function is executed and the time the `__stdcx` function completes. This has the same effect as inserting `__fence` built-in functions before and after the `__ldarx` built-in function and can inhibit compiler optimization of surrounding code (see “`__alignx`” on page 362 for a description of the `__fence` built-in function).

`__ldarx` is valid only in 64-bit mode.

Store functions

`__stdcx`, `__stwcx`

Purpose

Store Doubleword Conditional Indexed, Store Word Conditional Indexed

Stores the value specified by *val* into the memory location specified by *addr*.

Prototype

```
int __stdcx(volatile long* addr, long val);
```

```
int __stwcx(volatile int* addr, int val);
```

Parameters

addr

The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value

The value which is to be assigned to *addr*.

Return value

Returns 1 if the update of *addr* is successful and 0 if it is unsuccessful.

Usage

This function can be used with a preceding `__ldarx` (or `__ldarx`) built-in function to implement a read-modify-write on a specified memory location. The two built-in functions work together to ensure that if the store is successfully performed, no other processor or mechanism can modify the target doubleword between the time the `__ldarx` function is executed and the time the `__stdcx` function completes. This has the same effect as inserting `__fence` built-in functions before and after the `__stdcx` built-in function and can inhibit compiler optimization of surrounding code.

`__stdcx` is valid only in 64-bit mode.

Synchronization functions

`__eieio`, `__iospace_eieio`

Purpose

Enforce In-order Execution of Input/Output

Ensures that all I/O storage access instructions preceding the call to `__eieio` complete in main memory before I/O storage access instructions following the function call can execute.

Prototype

```
void __eieio (void);
```

```
void __iospace_eieio (void);
```

Usage

This function is useful for managing shared data instructions where the execution order of load/store access is significant. The function can provide the necessary functionality for controlling I/O stores without the cost to performance that can occur with other synchronization instructions.

`__isync`, `__iospace_sync`

Purpose

Instruction Synchronize

Waits for all previous instructions to complete and then discards any prefetched instructions, causing subsequent instructions to be fetched (or refetched) and executed in the context established by previous instructions.

Prototype

```
void __isync (void);
```

```
void __iospace_sync (void);
```

`__lwsync`, `__iospace_lwsync`

Purpose

Load Word Synchronize

Ensures that all store instructions preceding the call to `__lwsync` complete before any new instructions can be executed on the processor that executed the function. This allows you to synchronize between multiple processors with minimal performance impact, as `__lwsync` does not wait for confirmation from each processor.

Prototype

```
void __lwsync (void);  
  
void __iospace_lwsync (void);
```

__sync

Purpose

Synchronize

Ensures that all instructions preceding the function the call to `__sync` complete before any instructions following the function call can execute.

Prototype

```
void __sync (void);
```

Cache-related built-in functions (PPU only)

Cache-related built-in functions are grouped into the following categories:

- “Data cache functions”
- “Prefetch functions” on page 361

Data cache functions

__dcbf

Purpose

Data Cache Block Flush

Copies the contents of a modified block from the data cache to main memory and flushes the copy from the data cache.

Prototype

```
void __dcbf(const void* addr);
```

__dcbst

Purpose

Data Cache Block Store

Copies the contents of a modified block from the data cache to main memory.

Prototype

```
void __dcbst(const void* addr);
```

__dcbt

Purpose

Data Cache Block Touch

Loads the block of memory containing the specified address into the L1 data cache.

Prototype

```
void __dcbt (void* addr);
```

__dcbtst

Purpose

Data Cache Block Touch for Store

Fetches the block of memory containing the specified address into the data cache.

Prototype

```
void __dcbtst(void* addr);
```

__dcbz

Purpose

Data Cache Block set to Zero

Sets a cache line containing the specified address in the data cache to zero (0).

Prototype

```
void __dcbz (void* addr);
```

Prefetch functions

__prefetch_by_load

Purpose

Touches a memory location by using an explicit load.

Prototype

```
void __prefetch_by_load (const void*);
```

__prefetch_by_stream

Purpose

Touches a memory location by using an explicit stream.

Prototype

```
void __prefetch_by_stream (const int, const void*);
```

Block-related built-in functions

__bcopy

Purpose

Block copy

Prototype

```
void __bcopy (char*, char*, size_t);
```

Miscellaneous built-in functions

Miscellaneous functions are grouped into the following categories:

- “Optimization-related functions”
- “Move to/from register functions” on page 363
- “Memory-related functions” on page 365

Optimization-related functions

`__align_hint` (SPU only)

Purpose

Allows for optimizations such as automatic vectorization by informing the compiler that the data pointed to by *pointer* is aligned at a known compile-time offset.

Prototype

```
void __align_hint (const void* pointer, int base, int offset);
```

Parameters

base

Must be a constant integer with a value greater than zero and of a power of two.

offset

Must be less than *base* or zero.

Usage

To use this function, you must include the header file `spu_intrinsics.h` in your source.

`__alignx`

Purpose

Allows for optimizations such as automatic vectorization by informing the compiler that the data pointed to by *pointer* is aligned at a known compile-time offset.

Prototype

```
void __alignx (int alignment, const void* pointer);
```

Parameters

alignment

Must be a constant integer with a value greater than zero and of a power of two.

__builtin_expect

Purpose

Indicates that an expression is likely to evaluate to a specified value. The compiler may use this knowledge to direct optimizations.

Prototype

```
long __builtin_expect (long expression, long value);
```

Parameters

expression

Should be an integral-type expression.

value

For code targeting the PPU, *value* must be a constant literal. For code targeting the SPU, it can be an integral-type expression.

Usage

If the *expression* does not actually evaluate at run time to the predicted value, performance may suffer. Therefore, this built-in function should be used with caution.

__fence

Purpose

Acts as a barrier to compiler optimizations that involve code motion, or reordering of machine instructions. Compiler optimizations will not move machine instructions past the location of the `__fence` call.

Prototype

```
void __fence (void);
```

Examples

This function is useful to guarantee the ordering of instructions in the object code generated by the compiler when optimization is enabled.

Move to/from register functions

__mftb (PPU only)

Purpose

Move from Time Base

In 32-bit compilation mode, returns the lower word of the time base register. In 64-bit mode, returns the entire doubleword of the time base register.

Prototype

```
unsigned long __mftb (void);
```

Usage

In 32-bit mode, this function can be used in conjunction with the `__mftbu` built-in function to read the entire time base register. In 64-bit mode, the entire doubleword of the time base register is returned, so separate use of `__mftbu` is unnecessary.

It is recommended that you insert the `__fence` built-in function before and after the `__mftb` built-in function.

`__mftbu` (PPU only)

Purpose

Move from Time Base Upper

Returns the upper word of the time base register.

Prototype

```
unsigned int __mftbu (void);
```

Usage

In 32-bit mode you can use this function in conjunction with the `__mftb` built-in function to read the entire time base register.

It is recommended that you insert the `__fence` built-in function before and after the `__mftbu` built-in function.

`__mfmsr` (PPU only)

Purpose

Move from Machine State Register

Moves the contents of the machine state register (MSR) into bits 32 to 63 of the designated general-purpose register.

Prototype

```
unsigned long __mfmsr (void);
```

Usage

Execution of this instruction is privileged and restricted to supervisor mode only.

`__mfspr` (PPU only)

Purpose

Move from Special-Purpose Register

Returns the value of given special purpose register.

Prototype

```
unsigned long __mfspr (const int registerNumber);
```

Parameters

registerNumber

The number of the special purpose register whose value is to be returned. The *registerNumber* must be known at compile time.

__mtmsr (PPU only)

Purpose

Move to Machine State Register

Moves the contents of bits 32 to 63 of the designated GPR into the MSR.

Prototype

```
void __mtmsr (unsigned long);
```

Usage

Execution of this instruction is privileged and restricted to supervisor mode only.

__mtspr (PPU only)

Purpose

Move to Special-Purpose Register

Sets the value of a special purpose register.

Prototype

```
void __mtspr (const int registerNumber, unsigned long value);
```

Parameters

registerNumber

The number of the special purpose register whose value is to be set. The *registerNumber* must be known at compile time.

value

Must be known at compile time.

Memory-related functions

__alloca

Purpose

Allocates space for an object. The allocated space is put on the stack and freed when the calling function returns.

Prototype

```
void* __alloca (size_t size)
```

Parameters

size

An integer representing the amount of space to be allocated, measured in bytes.

__builtin_frame_address, __builtin_return_address

Purpose

Returns the address of the stack frame, or return address, of the current function, or of one of its callers.

Prototype

```
void* __builtin_frame_address (unsigned int level);
```

```
void* __builtin_return_address (unsigned int level);
```

Parameters

level

A constant literal indicating the number of frames to scan up the call stack. The *level* must range from 0 to 63. A value of 0 returns the frame or return address of the current function, a value of 1 returns the frame or return address of the caller of the current function and so on.

Return value

Returns 0 when the top of the stack is reached. Optimizations such as inlining may affect the expected return value by introducing extra stack frames or fewer stack frames than expected. If a function is inlined, the frame or return address corresponds to that of the function that is returned to.

Built-in functions for parallel processing

Use these built-in functions to obtain information about the parallel environment:

- “OpenMP built-in functions”

OpenMP built-in functions

Function definitions for the `omp_` functions can be found in the `omp.h` header file.

For complete information about OpenMP runtime library functions, refer to the OpenMP C/C++ Application Program Interface specification in www.openmp.org.

Related information

- “Environment variables for parallel processing” on page 23

omp_get_num_threads

Purpose

Returns the number of threads currently in the team executing the parallel region from which it is called.

Prototype

```
int omp_get_num_threads (void);
```

omp_set_num_threads

Purpose

Overrides the setting of the `OMP_NUM_THREADS` environment variable, and specifies the number of threads to use in parallel regions following this directive.

Prototype

```
void omp_set_num_threads (int num_threads);
```

Parameters

num_threads

Must be a positive integer.

Usage

If the `num_threads` clause is present, then for the parallel region it is applied to, it supersedes the number of threads requested by this function or the `OMP_NUM_THREADS` environment variable. Subsequent parallel regions are not affected by it.

omp_get_max_threads

Purpose

Returns the maximum value that can be returned by calls to `omp_get_num_threads`.

Prototype

```
int omp_get_max_threads (void);
```

omp_get_thread_num

Purpose

Returns the thread number, within its team, of the thread executing the function.

Prototype

```
int omp_get_thread_num (void);
```

Return value

The thread number lies between 0 and `omp_get_num_threads()-1`, inclusive. The master thread of the team is thread 0.

omp_get_num_procs

Purpose

Returns the maximum number of processors that could be assigned to the program.

Prototype

```
int omp_get_num_procs (void);
```

omp_in_parallel

Purpose

Returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, returns 0.

Prototype

```
int omp_in_parallel (void);
```

omp_set_dynamic

Purpose

Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.

Prototype

```
void omp_set_dynamic (int dynamic_threads);
```

omp_get_dynamic

Purpose

Returns non-zero if dynamic thread adjustments enabled and returns 0 otherwise.

Prototype

```
int omp_get_dynamic (void);
```

omp_set_nested

Purpose

Enables or disables nested parallelism.

Prototype

```
void omp_set_nested (int);
```

Return value

In the current implementation, nested parallel regions are always serialized. As a result, has no effect.

omp_get_nested

Purpose

Returns non-zero if nested parallelism is enabled and 0 if it is disabled.

Prototype

```
int omp_get_nested (void);
```

Return value

In the current implementation, nested parallel regions are always serialized. As a result, always returns 0.

omp_init_lock, omp_init_nest_lock

Purpose

Initializes the lock associated with the parameter *lock* for use in subsequent calls.

Prototype

```
void omp_init_lock (omp_lock_t *lock);  
  
void omp_init_nest_lock (omp_nest_lock_t *lock);
```

omp_destroy_lock, omp_destroy_nest_lock

Purpose

Ensures that the specified lock variable *lock* is uninitialized.

Prototype

```
void omp_destroy_lock (omp_lock_t *lock);  
  
void omp_destroy_nest_lock (omp_nest_lock_t *lock);
```

omp_set_lock, omp_set_nest_lock

Purpose

Blocks the thread executing the function until the specified lock is available and then sets the lock.

Prototype

```
void omp_set_lock (omp_lock_t * lock);  
  
void omp_set_nest_lock (omp_nest_lock_t * lock);
```

Usage

A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function.

omp_unset_lock, omp_unset_nest_lock

Purpose

Releases ownership of a lock.

Prototype

```
void omp_unset_lock (omp_lock_t * lock);  
  
void omp_unset_nest_lock (omp_nest_lock_t * lock);
```

omp_test_lock, omp_test_nest_lock

Purpose

Attempts to set a lock but does not block execution of the thread.

Prototype

```
int omp_test_lock (omp_lock_t * lock);  
  
int omp_test_nest_lock (omp_nest_lock_t * lock);
```

omp_get_wtime

Purpose

Returns the time elapsed from a fixed starting time.

Prototype

```
double omp_get_wtime (void);
```

Usage

The value of the fixed starting time is determined at the start of the current program, and remains constant throughout program execution.

omp_get_wtick

Purpose

Returns the number of seconds between clock ticks.

Prototype

```
double omp_get_wtick (void);
```

Usage

The value of the fixed starting time is determined at the start of the current program, and remains constant throughout program execution.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2008. All rights reserved.

Trademarks and service marks

IBM, the IBM logo, and `ibm.com` are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

- qatomic_updates compiler option 61
- qea32 compiler option 88
- qea64 compiler option 88
- qfdpr compiler option 96
- qnoatomic_updates compiler option 61
- qoptdebug compiler option 182
- qreport compiler option 202
- qsaveopt compiler option 209
- qsmp compiler option 214
- qswcache_size 233
- qversion compiler option 258

A

- alias 51
 - qalias compiler option 51
 - pragma disjoint 280
- alignment 53
 - qalign compiler option 53
 - pragma align 53
 - pragma pack 298
- alter program semantics 227
- appending macro definitions, preprocessed output 212
- architecture 57
 - q32 compiler option 49
 - q64 compiler option 49
 - qarch compiler option 57
 - qcache compiler option 67
 - qtune compiler option 250
 - architecture combination 251
 - macros 330
- arrays
 - padding 115

B

- basic example, described xi
- built-in functions 339
 - block-related 361
 - cache-related 360
 - fixed-point 339
 - floating-point 345
 - for parallel processing 366
 - miscellaneous 362
 - synchronization and atomic 352

C

- cleanpdf command 188
- compatibility
 - qabi_version compiler option 49
 - compatibility
 - options for compatibility 45
- compiler options 6
 - qatomic_updates 61
 - qea32 88
 - qea64 88

- compiler options (*continued*)
 - qnoatomic_updates 61
 - qswcache_size 233
 - performance optimization 41
 - resolving conflicts 9
 - specifying compiler options 6
 - command line 7
 - configuration file 8
 - source files 8
 - summary of command line options 31
- configuration 24
 - custom configuration files 24
 - ppugxlc, ppugxlc++, spugxlc, or spugxlc++ 27
 - specifying compiler options 8
- configuration file 94
- control of implicit timestamps 245
- control of transformations 227

D

- data types 56
 - qaltivec compiler option 56
- debug optimized code 182
- dynamic profiling environment variable 24

E

- effective address space support
 - qatomic_updates compiler option 61
 - qea32 compiler option 88
 - qea64 compiler option 88
 - qnoatomic_updates compiler option 61
 - qswcache_size compiler option 233
- environment variable 21
- environment variables 22
- scheduling algorithm environment variable 23
- XLSMPOPTS environment variable 23
- error checking and debugging 37
 - g compiler option 108
 - qcheck compiler option 70
 - qlinedebug compiler option 163
- exception handling
 - for floating point 103

F

- floating-point
 - exceptions 103

G

- GCC options 10

H

- high order transformation 115

I

- implicit timestamps, control of 245
- inlining 198
- interprocedural analysis (IPA) 132
- invocations 1
 - compiler or components 1
 - preprocessor 12
 - selecting 1
 - syntax 3

L

- language standards 148
 - qlanglvl compiler option 148
- large pages 160
- lib*.a library files 146
- lib*.so library files 146
- libraries
 - libraries
 - redistributable 14
 - XL C/C++ 14
- linker 13
 - invoking 13
- linking 13
 - options that control linking 44
 - order of linking 14
- listing 18, 211
 - qattr compiler option 62
 - qlist compiler option 164
 - qlistopt compiler option 165
 - qsource compiler option 218
 - qxref compiler option 265
 - options that control listings and messages 39

M

- machines, compiling for different types 57
- macro definitions, preprocessed output 212
- macros 325
 - independent of compiler options 337
 - related to architecture 330
 - related to compiler options 328
 - related to language features 331
 - related to the compiler 326
 - related to the platform 327
- maf suboption of -qfloat 230

O

- object output, implicit timestamps 245
- OMP_DYNAMIC environment variable 24

- OMP_NESTED environment variable 23
- OMP_NUM_THREADS environment variable 24
- OMP_SCHEDULE environment variable 23
- OpenMP 23
 - OpenMP environment variables 23
- optimization 41
 - O compiler option 179
 - qalias compiler option 51
 - qoptimize compiler option 179
 - controlling, using option_override pragma 296
 - loop optimization 41
 - qhot compiler option 115
 - qstrict_induction compiler option 231
 - options for performance optimization 41

P

- parallel processing 23
 - built-in functions 366
 - OpenMP environment variables 23
 - parallel processing pragmas 312
 - pragma directives 312
 - setting parallel processing environment variables 23
- performance 41
 - O compiler option 179
 - qalias compiler option 51
 - qoptimize compiler option 179
- platform, compiling for a specific type 57
- ppugxlc, ppugxlc++, spugxlc, and spugxlc++ utilities 10
- pragmas
 - priority 194
 - report 304
- priority pragma 194
- profile-directed feedback (PDF) 187
 - qpdf1 compiler option 187
 - qpdf2 compiler option 187
- profiling 183
 - qpdf1 compiler option 187
 - qpdf2 compiler option 187
 - environment variable 24

R

- report
 - pragma 304
- resetpdf command 188
- rrm suboption of -qfloat 230

S

- shared objects 176
 - qmshrobj 176
- shared-memory parallelism (SMP)
 - qsmp compiler option 214
- SIGTRAP signal 103

T

- target machine, compiling for 57
- templates 239
 - qtempinc compiler option 239
 - qtemplaterecompile compiler option 241
 - qtemplateregistry compiler option 242
 - qtempmax compiler option 243
 - qtmplinst compiler option 247
 - qtmplparse compiler option 248
 - pragma define 279
 - pragma do_not_instantiate 281
 - pragma implementation 287
 - pragma instantiate 279
- transformations, control of 227
- tuning 250
 - qarch compiler option 250
 - qtune compiler option 250

V

- vector data types 56
 - qaltivec compiler option 56
- vector processing 94
 - qaltivec compiler option 56
- virtual function table (VFT) 85
 - qdump_class_hierarchy 85
 - pragma hashome 285, 288



Program Number: 5724-T42 & 5724-T43

Printed in USA

SC23-8570-00

