IBM XL C/C++ for Blue Gene/Q, V12.1

Compiler Reference

Version 12.1

GC14-7363-00

IBM XL C/C++ for Blue Gene/Q, V12.1

Compiler Reference

Version 12.1

GC14-7363-00

Note
Before using this information and the product it supports, read the information in "Notices" on page 513.

First edition

This edition applies to IBM XL C/C++ for Blue Gene/Q, V12.1 (Program 5799-AG1) 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.

Contents

About this information

This information is a reference for the IBM® XL C/C++ for Blue Gene®/Q, V12.1 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.

Throughout this information, the **bgxlc** and **bgxlc++** command invocations are used to describe the actions of the compiler. You can, however, substitute other forms of the compiler invocation command if your particular environment requires it, and compiler option usage will remain the same unless otherwise specified.

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 **gxlc** and **gxlc++**; 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 **gxlc** and **gxlc++** option mappings.

Chapter 3, “Compiler options reference,” on page 43 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 303 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, including OpenMP directives.

Chapter 5, “Compiler predefined macros,” on page 381 provides a list of compiler macros according to category.

Chapter 6, “Compiler built-in functions,” on page 395 contains individual descriptions of XL C/C++ built-in functions for Power® architectures, categorized by their functionality.

Conventions

Typographical conventions

The following table shows the typographical conventions used in the IBM XL C/C++ for Blue Gene/Q, V12.1 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **bgxlc** and **bgxlC** (**bgxlc++**), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
italics	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 *size* parameter if you return more than the *size* requested.
underlining	The default setting of a parameter of a compiler option or directive.	nomaf | maf
`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: `bgxlc myprogram.c -O3`.

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.
C1X, or C1X begins   C1X ends	The text describes a feature that is introduced into standard C as part of C1X.
C++0x, or C++0x begins   C++0x ends	The text describes a feature that is introduced into standard C++ as part of C++0x.

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.

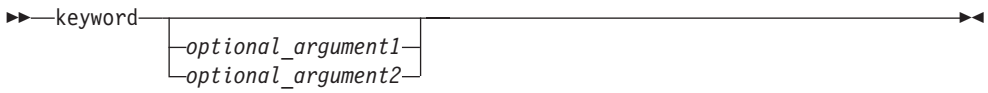
  Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the |— symbol and end with the —| symbol.
- Required items are shown on the horizontal line (the main path):

```
▸▸—keyword—required_argument—————————————————————————————▸◂
```

- 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.



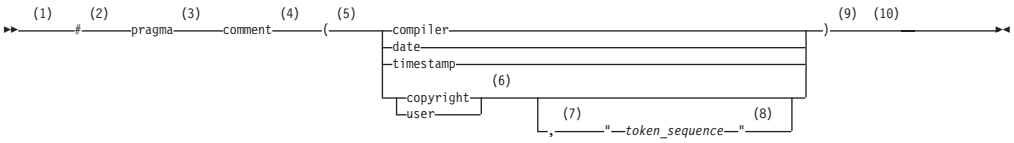
- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

- 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.

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 Blue Gene/Q, V12.1 Installation Guide*.

- Information center

  The information center of searchable HTML files can be launched on a network and accessed remotely or locally. Instructions for installing and accessing the online information center are provided in the *IBM XL C/C++ for Blue Gene/Q, V12.1 Installation Guide*.

  The information center of searchable HTML files is viewable on the web at http://pic.dhe.ibm.com/infocenter/compbg/v121v141/index.jsp.

- PDF documents

  PDF documents are located by default in the /opt/ibmcmp/vacpp/bg/12.1/doc/en_US/pdf/ directory. The PDF files are also available on the web at http://www.ibm.com/software/awdtools/xlcpp/features/bg/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
IBM XL C/C++ for Blue Gene/Q, V12.1 Installation Guide, GC14-7362-00	install.pdf	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C/C++ for Blue Gene/Q, V12.1, GC14-7361-00	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.
IBM XL C/C++ for Blue Gene/Q, V12.1 Compiler Reference, GC14-7363-00	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions, including those used for parallel processing.
IBM XL C/C++ for Blue Gene/Q, V12.1 Language Reference, GC14-7364-00	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.
IBM XL C/C++ for Blue Gene/Q, V12.1 Optimization and Programming Guide, SC14-7365-00	proguide.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization and parallelization, 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 website at http://www.adobe.com.

More information related to XL C/C++ including IBM Redbooks® publications, white papers, tutorials, and other articles, is available on the web at:

http://www.ibm.com/software/awdtools/xlcpp/features/bg/library/

For more information about boosting performance, productivity, and portability, see the C/C++ café at http://www.ibm.com/software/rational/cafe/community/ccpp.

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.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- *OpenMP Application Program Interface Version 3.1*, available at http://www.openmp.org

Other IBM information

- *Blue Gene/Q Hardware Overview and Installation Planning, SG24-7872*, available at http://www.redbooks.ibm.com/redpieces/abstracts/sg247872.html?Open
- *Blue Gene/Q Hardware Installation and Maintenance Guide, SG24-7974*, available at http://www.redbooks.ibm.com/redpieces/abstracts/sg247974.html?Open
- *Blue Gene/Q High Availability Service Node, REDP-4657*, available at http://www.redbooks.ibm.com/redpieces/abstracts/redp4657.html?Open
- *Blue Gene/Q System Administration, SG24-7869*, available at http://www.redbooks.ibm.com/redpieces/abstracts/sg247869.html?Open
- *Blue Gene/Q Application Development, SG24-7948*, available at http://www.redbooks.ibm.com/redpieces/abstracts/sg247948.html?Open
- *Blue Gene/Q Code Development and Tools Interface, REDP-4659*, available at http://www.redbooks.ibm.com/redpieces/abstracts/redp4659.html?Open

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/features/bg/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 email 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/features/bg/.

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 email 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 call 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

To compile a source program, use any of the available XL C/C++ for Blue Gene/Q compiler invocation commands. The **bg**-prefixed invocation commands on the Blue Gene/Q Front End node (RHEL 6.2) are for cross-compiling applications for use on the Blue Gene/Q Compute node. The invocation commands that are not prefixed with **bg** create executable programs targeted for RHEL 6.2 on POWER® platforms, and are provided only for testing and debugging purposes. For the development of applications targeted for the Front End node, IBM provides the IBM XL C/C++ for Linux, V12.1 product. As well, only the compiler options which are supported by the **bg** cross-compiler commands are supported when using these compiler invocations to create executable files for Blue Gene/Q.

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 **bgxlc** command to compile your C source files, and the **bgxlc++** command to compile C++ source files. Use **bgxlc++** to link if you have both C and C++ object files.

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 2.

Note: For each invocation command, the compiler configuration file defines default option settings and, in some cases, macros; for information about the defaults implied by a particular invocation, see the /opt/ibmcmp/vac/bg/12.1/etc/vac.cfg.*$OSRelease.gcc$gccVersion* file for your system. For example, /opt/ibmcmp/vac/bg/12.1/etc/vac.cfg.sles11.gcc432 or /opt/ibmcmp/vac/bg/12.1/etc/vac.cfg.rhel6.2.gcc446.

Table 4. Compiler invocations

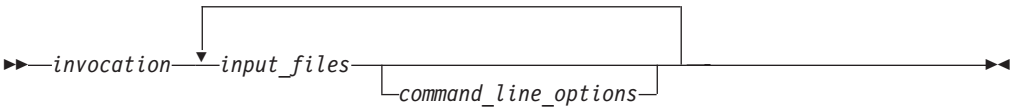
Basic invocations	Description	Equivalent special invocations
bgxlc	Invokes the compiler for C source files. This command supports all of the ISO C99 standard features, and most IBM language extensions. This invocation is recommended for all applications.	bgxlc_r
bgc99	Invokes the compiler for C source files. This command supports all ISO C99 language features, but does not support IBM language extensions. Use this invocation for strict conformance to the C99 standard.	bgc99_r
bgc89	Invokes the compiler for C source files. This command supports all ANSI C89 language features, but does not support IBM language extensions. Use this invocation for strict conformance to the C89 standard.	bgc89_r
bgcc	Invokes the compiler for C source files. 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.	bgcc_r
bgxlc++, bgxlC	Invokes the compiler for C++ source files. 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 **-+** compiler option, are compiled as C language source code.	bgxlc++_r, bgxlC_r

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:



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 are displayed. 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 does not run and temporary object files are removed.

By default, the compiler preprocesses and compiles all the specified source files. Although you usually want to use this default, you can use the compiler to preprocess the source file without compiling; see "Preprocessing" on page 10 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 **-qsourcetype=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 **-qsourcetype=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
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
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. The XL C/C++ compiler 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 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 in the same command line compiler invocation, the subsequent option in the invocation 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, command line or source program override compiler default settings.

Option conflicts that do not follow this priority sequence are described in "Resolving conflicting compiler options" on page 8.

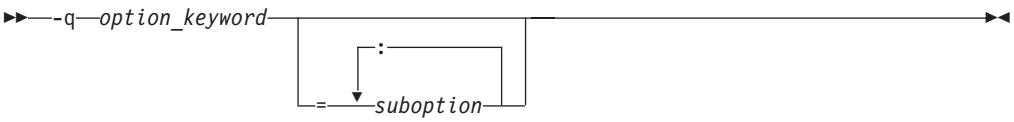
Specifying compiler options on the command line

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 8.

There are two kinds of command-line options:

- **-q***option_keyword* (compiler-specific)
- Flag options

-q options



Command-line options in the **-q***option_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 used 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:

```
bgxlc -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 **-q***option_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 **-q***option_keyword* before or after the file name. For example:

```
bgxlc -qLIST -qfloat=nomaf file.c
bgxlc 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 **-q***option*. If the option permits more than one suboption, a colon (**:**) must separate each suboption from the next. For example:

```
bgxlc -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:

```
bgxlc 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:

```
bgxlc -Ocv file.c
```

has the same effect as:

```
bgxlc -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:

```
bgxlc -Ovo test test.c
```

has the same effect as:

```
bgxlc -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/vac/bg/12.1/etc/vac.cfg.*$OSRelease*.*gcc$gccVersion*. For example, /opt/ibmcmp/vac/bg/12.1/etc/vac.cfg.sles11.gcc432 or /opt/ibmcmp/vac/bg/12.1/etc/vac.cfg.rhel6.2.gcc446) 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 39.

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, you can have several ways to specify the syntax of the pragmas:

- Using **#pragma options** ***option_name*** syntax — You can use command-line options with 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. For details, see "#pragma options" on page 334.
- 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 59, 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 303.

Other pragmas do not have equivalent command-line options; these are described in detail throughout Chapter 4, "Compiler pragmas reference," on page 303.

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 **-I***directory* and **-L***directory* 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 preceding rules. The following table summarizes exceptions and how the compiler handles conflicts between them. Rules for resolving conflicts between compiler mode and architecture-specific options are discussed in "Specifying compiler options for architecture-specific compilation" on page 9.

Option	Conflicting options	Resolution
-qalias=allptrs	-qalias=noansi	-qalias=noansi

Option	Conflicting options	Resolution
-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

Specifying compiler options for architecture-specific compilation

You can use the **-q64**, **-qarch**, and **-qtune** compiler options to optimize the output of the compiler to suit:

- The broadest possible selection of target processors
- A range of processors within a given processor architecture family
- A single specific processor

Generally speaking, the options do the following:

- **-q64** selects 64-bit execution mode.
- **-qarch** selects the general family processor architecture for which instruction code should be generated. Certain **-qarch** settings produce code that will run *only* on systems that support *all* of the instructions generated by the compiler in response to a chosen **-qarch** setting.
- **-qtune** selects the specific processor for which compiler output is optimized. Some **-qtune** settings can also be specified as **-qarch** options, in which case they do not also need to be specified as a **-qtune** option. The **-qtune** option influences only the performance of the code when running on a particular system but does not determine where the code will run.

The compiler evaluates compiler options in the following order, with the last allowable one found determining the compiler mode:

1. Internal default (64-bit mode)
2. Configuration file settings
3. Command line compiler options (**-q64**, **-qarch**, **-qtune**)
4. Source file statements (**#pragma options tune=***suboption*)

The compilation mode actually used by the compiler depends on a combination of the settings of the **-q64**, **-qarch**, and **-qtune** compiler options, subject to the following conditions:

- *Compiler mode* is set according to the last-found instance of the **-q64** compiler option.
- *Architecture target* is set according to the last-found instance of the **-qarch** compiler option, provided that the specified **-qarch** setting is compatible with the *compiler mode* setting. If the **-qarch** option is not set, the compiler sets **-qarch** to the appropriate default based on the effective compiler mode setting. See "-qarch" on page 69 for details.
- Tuning of the architecture target is set according to the last-found instance of the **-qtune** compiler option, provided that the **-qtune** setting is compatible with the *architecture target* and *compiler mode* settings. If the **-qtune** option is not set, the compiler assumes a default **-qtune** setting according to the **-qarch** setting in use. If **-qarch** is not specified, the compiler sets **-qtune** to the appropriate default based on the effective **-qarch** as selected by default based on the effective compiler mode setting.

Allowable combinations of these options are found in "-qtune" on page 284.

The following list describes possible option conflicts and compiler resolution of these conflicts:

- **-q64** setting is incompatible with user-selected **-qarch** option.

  Resolution: **-q64** setting overrides the **-qarch** option; compiler issues a warning message, sets **-qarch** to its default setting, and sets the **-qtune** option accordingly to its default value.
- **-qarch** option is incompatible with user-selected **-qtune** option.

  Resolution: Compiler issues a warning message, and sets **-qtune** to the **-qarch** setting's default **-qtune** value.
- Selected **-qarch** or **-qtune** options are not known to the compiler.

  Resolution: Compiler issues a warning message, sets **-qarch** and **-qtune** to their default settings.

Related information
- "-qarch" on page 69
- "-qtune" on page 284
- "-q64" on page 62

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 100	Preprocesses the source files and writes the output to standard output. By default, `#line` directives are generated.
"-P" on page 215	Preprocesses the source files and creates an intermediary file with a .i file name suffix for each source file. By default, `#line` directives are not generated.
"-qppline" on page 222	Toggles on and off the generation of `#line` directives for the **-E** and **-P** options.
"-C, -C!" on page 78	Preserves comments in preprocessed output.
"-D" on page 93	Defines a macro name from the command line, as if in a `#define` directive.
"-U" on page 285	Undefines a macro name defined by the compiler or by the **-D** option.
"-qshowmacros" on page 242	Emits macro definitions to preprocessed output.

Directory search sequence for include files

The XL C/C++ compiler 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 displays on the command line.

4. All header files: C 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/vac/bg/12.1/include/, but the search path can be changed with the **-qc_stdinc** compiler option. C++ 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/vacpp/bg/12.1/include/, but the search path can be changed with the **-qcpp_stdinc** compiler option.[2]
5. All header files: C The compiler searches the standard directory for the system headers. The default directory for these headers is specified in the compiler configuration file. The default location for these headers is specified in the compiler configuration file. This location is set during installation, but the search path can be changed with the **-qgcc_c_stdinc** option. C++ The compiler searches the standard directory for the system headers. The default directory for these headers is specified in the compiler configuration file. The default location for these headers is specified in the compiler configuration file. This location is set during installation, 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 133
- "-qc_stdinc (C only)" on page 90
- "-qcpp_stdinc (C++ only)" on page 91
- "-qgcc_c_stdinc (C only)" on page 124
- "-qgcc_cpp_stdinc (C++ only)" on page 125
- "-qidirfirst" on page 134
- "-qinclude" on page 137
- "-qstdinc" on page 260

Linking

On Blue Gene/Q platforms, static linking is enabled by default. To link dynamically, use the **-qnostaticlink** compiler option.

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:

```
bgxlc 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 **ld** 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 57.

Note: If you want to use a nondefault linker, you can use either of the following options:

- Use **-t** and **-B** to specify the nondefault linker, for example,
  `-tl -B`*`linker_path`*
- Customize the configuration file of the compiler to use the nondefault linker. For more information about how to customize the configuration file, see Using custom compiler configuration files and Creating custom configuration files.

Related information

- "-qmkshrobj" on page 205
- -qnostaticlink

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 57
- "Redistributable libraries"

Redistributable libraries

If you build your application using XL C/C++, it might 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, you must do one of the following:

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

  http://www.ibm.com/software/awdtools/xlcpp/features/bg/support/

For information about the licensing requirements related to the distribution of these packages see the LicAgree.pdf file on the CD.

Table 6. Redistributable libraries

Package name	Libraries (and default installation path)	Description
vac.lib	/opt/ibmcmp/vac/bg/V12.1/lib/libxl.a /opt/ibmcmp/vac/bg/V12.1/lib64/libxl.a /opt/ibmcmp/vac/bg/V12.1/lib/libxlopt.a /opt/ibmcmp/vac/bg/V12.1/lib64/libxlopt.a	XL C compiler libraries
vacpp.rte	/opt/ibmcmp/vac/bg/V12.1/lib/libibmc++.so.1 /opt/ibmcmp/vac/bg/V12.1/lib64/libibmc++.so.1	XL C++ runtime libraries
xlsmp.rte	/opt/ibmcmp/lib/libxlomp_ser.so.1 /opt/ibmcmp/lib/libxlsmp.so.1 /opt/ibmcmp/lib64/libxlomp_ser.so.1 /opt/ibmcmp/lib64/libxlsmp.so.1	SMP (OMP) runtime libraries
xlsmp.msg.rte	/opt/ibmcmp/msg/en_US/smprt.cat /opt/ibmcmp/msg/en_US.utf8/smprt.cat	SMP message catalogs (English)

Compiler messages and listings

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

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, or to a listing file if you compile with the **-qsource** option. These diagnostic messages are specific to the C or C++ language.

C 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 251
- "-qsrcmsg (C only)" on page 254
- "-qflag" on page 107
- "-w" on page 294
- "-qinfo" on page 139

Compiler message format

Diagnostic messages have the following format:

"*file*", line *line_number.column_number*: 15*dd-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.

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

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

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

Message severity levels and compiler response

The XL C/C++ compiler uses a multilevel 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. You can adjust the default compiler response by using any of the following options:

- **-qhalt** halts the compilation phase at a lower severity level than the default
- **-qmaxerr** halts the compilation phase as soon as a specific number of errors at a specific severity level is reached
- **-qhaltonmsg** halts 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.
C 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.
S	Severe error	Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct: • 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 compile-time error, the message should be reported to your IBM service representative.
C U	Unrecoverable error	The compiler halts. An internal compile-time error has occurred. The message should be reported to your IBM service representative.

Related information
- "-qhalt" on page 127
- "-qmaxerr" on page 199
- "-qhaltonmsg" on page 128
- 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.
- 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 can also be displayed for runtime errors.

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
- -qreport

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, release, 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 is displayed 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 about 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 has a file number to indicate which file the line came from.

Transformation report section

If the **-qreport** option is in effect, this section displays pseudo code that corresponds to the original source code, so that you can see parallelization and loop transformations that the **-qhot** or **-qsmp** option has generated. This section of the report also shows additional loop transformation and parallelization information about loop nests if you compile with **-qsmp** and **-qhot=level=2**.

This section also reports the number of streams created for a given loop and the location of data prefetch instructions inserted by the compiler. To generate information about data prefetch insertion locations, use the optimization level of **-qhot**, **-O3 -qhot**, **-O4** or **-O5** together with **-qreport**.

Data reorganization section

Displays data reorganization messages for program variable data during the IPA link pass when **-qreport** is used with **-qipa=level=2** or **-O5**. Reorganization information includes:

- array splitting
- array transposing
- memory allocation merging
- array interleaving
- array coalescing

Compilation epilogue section

Displays a summary of the diagnostic messages by severity level, the number of source lines read, and whether the compilation was successful.

Object section

If you use the **-qlist** option, lists the object code generated by the compiler. 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

Message catalog errors

Before the compiler can compile your program, the message catalogs must be installed and the environment variables *LANG* and *NLSPATH* must be set to a language for which the message catalog has been installed.

If you see the following message during compilation, the appropriate message catalog cannot be opened:

```
Error occurred while initializing the message system in
file: message_file
```

where *message_file* is the name of the message catalog that the compiler cannot open. This message is issued in English only.

You must then verify that the message catalogs and the environment variables are in place and correct. If the message catalog or environment variables are not correct, compilation can continue, but diagnostic messages are suppressed and the following message is issued instead:

```
No message text for message_number
```

where *message_number* is the compiler internal message number. This message is issued in English only.

To determine which message catalogs are installed on your system, assuming that you have installed the compiler to the default location, you can list all of the file names for the catalogs by the following command:

```
ls /opt/ibmcmp/vacpp/bg/12.1/msg/$LANG/*.cat
```

where *LANG* is the environment variable on your system that specifies the system locale.

The compiler calls the message catalogs for **en_US** by default if *LANG* is not set correctly.

For more information about the *NLSPATH* and *LANG* environment variables, see your operating system documentation.

Paging space errors during compilation

If the operating system runs low on paging space during a compilation, the compiler issues the following message:

```
1501-229 Compilation ended due to lack of space.
```

To minimize paging-space problems, do any of the following and recompile your program:

- Reduce the size of your program by splitting it into two or more source files
- Compile your program without optimization
- Reduce the number of processes competing for system paging space
- Increase the system paging space

For more information about paging space and how to allocate it, see your operating system documentation.

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 might 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, including those used for parallel processing.
- 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 other settings. Information on customizing the configuration file is provided in "Using custom compiler configuration files" on page 39.

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" on page 22
- "Runtime environment variables" on page 22

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 Blue Gene/Q system. With the exception of *LANG* and *NLSPATH*, which must be set if you are using a locale other than the default en_US, all of these variables are optional.

LANG
Specifies the locale for your operating system. The default locale used by the compiler for messages and help files is United States English, en_US, but the compiler supports other locales. For a list of these, see *National language support* in the *XL C/C++ Installation Guide*. For more information on setting the *LANG* environment variable to use an alternate locale, see your operating system documentation.

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 229.

NLSPATH
Specifies the directory search path for finding the compiler message and help files. You only need to set this environment variable if the national language to be used for the compiler message and help files is not English. For information on setting the *NLSPATH*, see *Enabling the XL C/C++ error messages* in the *XL C/C++ Installation Guide*.

PATH Specifies the directory search path for the executable files of the compiler. Executables are in /opt/ibmcmp/vacpp/bg/12.1/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 39.

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.

Environment variables for parallel processing

The XLSMPOPTS environment variable sets options for program run time using loop parallelization. Suboptions for the XLSMPOPTS environment variables are discussed in detail in "XLSMPOPTS."

If you are using OpenMP, transactional memory, or thread-level speculative execution constructs for parallelization, you can also specify runtime options using the corresponding environment variables, as discussed in the following sections.

When runtime options specified by OMP and XLSMPOPTS environment variables conflict, OMP options will prevail.

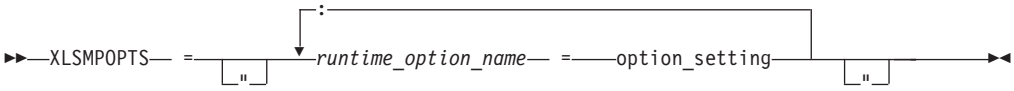
Note: You must use a threadsafe compiler mode invocations when compiling parallelized program code.

Related information

- "Pragma directives for parallel processing" on page 355
- "Built-in functions for parallel processing" on page 491

XLSMPOPTS

Runtime options affecting parallel processing can be specified with the XLSMPOPTS environment variable. This environment variable must be set before you run an application, and uses basic syntax of the form:



You can specify option names and settings in uppercase or lowercase. You can add blanks before and after the colons and equal signs to improve readability. However, if the XLSMPOPTS option string contains imbedded blanks, you must enclose the entire option string in double quotation marks (").

For example, to have a program run time create 4 threads and use dynamic scheduling with chunk size of 5, you would set the XLSMPOPTS environment variable as shown below:

```
XLSMPOPTS=PARTHDS=4:SCHEDULE=DYNAMIC=5
```

The following are the available runtime option settings for the XLSMPOPTS environment variable:

Scheduling options are as follows:

schedule
Specifies the type of scheduling algorithms and chunk size (*n*) that are used for loops to which no other scheduling algorithm has been explicitly assigned in the source code.

Work is assigned to threads in a different manner, depending on the scheduling type and chunk size used. Choosing chunking granularity is a

tradeoff between overhead and load balancing. The syntax for this option is **schedule**=*suboption*, where the suboptions are defined as follows:

affinity[=*n***]**
: The iterations of a loop are initially divided into *n* partitions, containing **ceiling**(*number_of_iterations*/*number_of_threads*) iterations. Each partition is initially assigned to a thread and is then further subdivided into chunks that each contain *n* iterations. If *n* is not specified, then the chunks consist of **ceiling**(*number_of_iterations_left_in_partition* / 2) loop iterations.

  When a thread becomes free, it takes the next chunk from its initially assigned partition. If there are no more chunks in that partition, then the thread takes the next available chunk from a partition initially assigned to another thread.

  The work in a partition initially assigned to a sleeping thread will be completed by threads that are active.

  The **affinity** scheduling type does not appear in the OpenMP API standard.

dynamic[=*n***]**
: The iterations of a loop are divided into chunks containing *n* iterations each. If *n* is not specified, then the chunks consist of **ceiling**(*number_of_iterations*/*number_of_threads*) iterations.

  Active threads are assigned these chunks on a "first-come, first-do" basis. Chunks of the remaining work are assigned to available threads until all work has been assigned.

  If a thread is asleep, its assigned work will be taken over by an active thread once that thread becomes available.

guided[=*n***]**
: The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of *n* loop iterations is reached. If *n* is not specified, the default value for *n* is 1 iteration.

  Active threads are assigned chunks on a "first-come, first-do" basis. The first chunk contains **ceiling**(*number_of_iterations*/*number_of_threads*) iterations. Subsequent chunks consist of **ceiling**(*number_of_iterations_left* / *number_of_threads*) iterations.

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 integral assignment expression of value 1 or greater.

Specifying **schedule** with no suboption is equivalent to **schedule=runtime**.

Parallel environment options are as follows:

parthds=*num*
Specifies the number of threads (*num*) requested, which is usually equivalent to the number of processors available on the system.

Some applications cannot use more threads than the maximum number of processors available. Other applications can experience significant performance improvements if they use more threads than there are processors. This option gives you full control over the number of user threads used to run your program.

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

usrthds=*num*
Specifies the maximum number of threads (*num*) that you expect your code will explicitly create if the code does explicit thread creation. The default value for *num* is 0.

stack=*num*
Specifies the largest amount of space in bytes (*num*) that a thread's stack needs. The default value for *num* is 4194304.

Set *num* so it is within the acceptable upper limit. *num* can be up to the limit imposed by system resources or the stack size ulimit, whichever is smaller. An application that exceeds the upper limit may cause a segmentation fault.

stackcheck[=*num***]**
When the **-qsmp=stackcheck** is in effect, enables stack overflow checking for slave threads at runtime. *num* is the size of the stack in bytes; when the remaining stack size is less than this value, a runtime warning message is issued. If you do not specify a value for *num*, the default value is 4096 bytes. Note that this option only has an effect when the **-qsmp=stackcheck** has also been specified at compile time. See "-qsmp" on page 247 for more information.

startproc=*cpu_id*
Enables thread binding and specifies the *cpu_id* to which the first thread binds. If the value provided is outside the range of available processors, a warning message is issued and no threads are bound.

procs=cpu_id*[,cpu_id,...]*
Enables thread binding and specifies a list of *cpu_id* to which the threads are bound. If the number of CPU IDs specified is less than the number of threads used by the program, the remaining threads are not bound.

stride=*num*
Specifies the increment used to determine the *cpu_id* to which subsequent threads bind. *num* must be greater than or equal to 1. If the value provided causes a thread to bind to a CPU outside the range of available processors, a warning message is issued and no threads are bound.

Performance tuning options are as follows:

spins=*num*
Specifies the number of loop spins, or iterations, before a yield occurs.

When a thread completes its work, the thread continues executing in a tight loop looking for new work. One complete scan of the work queue is done during each busy-wait state. An extended busy-wait state can make a particular application highly responsive, but can also harm the overall

responsiveness of the system unless the thread is given instructions to periodically scan for and yield to requests from other applications.

A complete busy-wait state for benchmarking purposes can be forced by setting both **spins** and **yields** to 0.

The default value for *num* is 100.

yields=*num*
Specifies the number of yields before a sleep occurs.

When a thread sleeps, it completely suspends execution until another thread signals that there is work to do. This provides better system utilization, but also adds extra system overhead for the application.

The default value for *num* is 100.

delays=*num*
Specifies a period of do-nothing delay time between each scan of the work queue. Each unit of delay is achieved by running a single no-memory-access delay loop.

The default value for *num* is 500.

Dynamic profiling options are as follows:

profilefreq=*n*
Specifies the frequency with which a loop should be revisited by the dynamic profiler to determine its appropriateness for parallel or serial execution.The runtime library uses dynamic profiling to dynamically tune the performance of automatically parallelized loops. Dynamic profiling gathers information about loop running times to determine if the loop should be run sequentially or in parallel the next time through. Threshold running times are set by the **parthreshold** and **seqthreshold** dynamic profiling options, described below.

The allowed values for this option are the numbers from 0 to 32. If *num* is 0, all profiling is turned off, and overheads that occur because of profiling will not occur. If *num* is greater than 0, running time of the loop is monitored once every *num* times through the loop. The default for *num* is 16. Values of *num* exceeding 32 are changed to 32.

It is important to note that dynamic profiling is not applicable to user-specified parallel loops.

parthreshold=*num*
Specifies the time, in milliseconds, below which each loop must execute serially. If you set *num* to 0, every loop that has been parallelized by the compiler will execute in parallel. The default setting is 0.2 milliseconds, meaning that if a loop requires fewer than 0.2 milliseconds to execute in parallel, it should be serialized.

Typically, *num* is set to be equal to the parallelization overhead. If the computation in a parallelized loop is very small and the time taken to execute these loops is spent primarily in the setting up of parallelization, these loops should be executed sequentially for better performance.

seqthreshold=*num*
Specifies the time, in milliseconds, beyond which a loop that was previously serialized by the dynamic profiler should revert to being a parallel loop. The default setting is 5 milliseconds, meaning that if a loop requires more than 5 milliseconds to execute serially, it should be parallelized.

seqthreshold acts as the reverse of **parthreshold**.

BG_SMP_FAST_WAKEUP

The BG_SMP_FAST_WAKEUP environment variable enables or disables the fast wake-up support, which allows the SMP runtime to use the hardware mechanism that puts waiting threads to sleep and wakes them up efficiently.

The default value is **NO**. The value is not case sensitive.

BG_SMP_FAST_WAKEUP=YES enables the fast wake-up support.

When a thread is waiting for work or spinning at a barrier, it puts itself to sleep so that it does not consume processing resources. Due to the Blue Gene/Q threading model, the sleeping cannot be interrupted. This causes an issue when the hardware threads are oversubscribed; that is, multiple user threads are bound to the same hardware thread. This issue might cause deadlocks between user threads, in particular between OpenMP threads in the SMP runtime.

To avoid oversubscription of hardware threads when the fast wake-up support is enabled, you must also set the OMP_PROC_BIND environment variable to TRUE. This setting ensures that each user thread created by the SMP runtime is bound to a different hardware thread.

Related information

- OMP_PROC_BIND

BG_SPEC_SCRUB_CYCLE

The BG_SPEC_SCRUB_CYCLE environment variable sets the L2 background scrub cycle, which determines how quick a speculation ID can be reclaimed by the L2 and used again for speculation.

The default value is 66. The valid value range is 6-100.

Setting the scrub cycle low means the L2 needs to perform scrub more frequently.

The interval defines the period in 800MHz cycles (two processor cycles) for directory look-ups of the scrub process. For the default setting, the L2 examines one of the 1024 sets every 132 processor cycles, resulting in a complete cache scrub every 135168 processor cycles.

After a speculation ID has been used and committed or recycled, it takes at most 135168 cycles until the ID is available again for allocation. It can take less if all sets that were accessed with the ID are visited by either core memory accesses or the scrub earlier.

For example, programming the interval to 6 changes the duration for a full cache scrub from 135168 to 12288 processor cycles.

Environment variables for OpenMP

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

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

For information about the OpenMP specification, see: http://www.openmp.org.

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

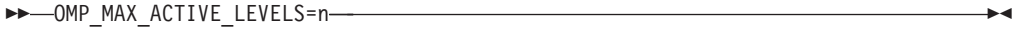
If it is set to TRUE, the number of threads available for executing parallel regions can be adjusted at run time to make the best use of system resources. For more information, see the description for **profilefreq=***num* in "XLSMPOPTS" on page 23.

If it is set to FALSE, dynamic adjustment is disabled.

The default setting is TRUE.

OMP_MAX_ACTIVE_LEVELS:

Use OMP_MAX_ACTIVE_LEVELS to set the *max-active-levels-var* internal control variable. This controls the maximum number of active nested parallel regions. In programs where nested parallelism is disabled, the initial value should be 1. In programs where nested parallelism is enabled, the initial value should be greater than 1. The function **omp_get_max_active_levels** can be used to retrieve this value at run time. The value for OMP_MAX_ACTIVE_LEVELS is a positive integer. If a positive integer is not specified, the default value for *max-active-levels-var* is set by the runtime.



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 might employ additional threads as available. However, no new team is created to run nested parallel regions.

The default value for OMP_NESTED is FALSE.

OMP_NUM_THREADS: The OMP_NUM_THREADS environment variable specifies the number of threads to use for parallel regions.

The syntax of the environment variable is as follows:

num_list
A list of one or more positive integer values separated by commas.

If you do not set OMP_NUM_THREADS, the number of processors available is the default value to form a new team for the first encountered parallel construct. By default, any nested parallel constructs are run by one thread.

If *num_list* contains a single value, dynamic adjustment of the number of threads is enabled (**OMP_DYNAMIC** is set to true), and a parallel construct without a **num_threads** clause is encountered, the value is the maximum number of threads that can be used to form a new team for the encountered parallel construct.

If *num_list* contains a single value, dynamic adjustment of the number of threads is not enabled (**OMP_DYNAMIC** is set to false), and a parallel construct without a **num_threads** clause is encountered, the value is the exact number of threads that can be used to form a new team for the encountered parallel construct.

If *num_list* contains multiple values, dynamic adjustment of the number of threads is enabled (**OMP_DYNAMIC** is set to true), a parallel construct without a **num_threads** clause is encountered, the first value is the maximum number of threads that can be used to form a new team for the encountered parallel construct. After the encountered construct is entered, the first value is removed and the remaining values form a new *num_list*. The new *num_list* is in turn used in the same way for any closely nested parallel constructs inside the encountered parallel construct.

If *num_list* contains multiple values, dynamic adjustment of the number of threads is not enabled (**OMP_DYNAMIC** is set to false), and a parallel construct without a **num_threads** clause is encountered, the first value is the exact number of threads that can be used to form a new team for the encountered parallel construct. After the encountered construct is entered, the first value is removed and the remaining values form a new *num_list*. The new *num_list* is in turn used in the same way for any closely nested parallel constructs inside the encountered parallel construct.

Note: If the number of parallel regions is equal to or greater than the number of values in *num_list*, the **omp_get_max_threads** function returns the last value of *num_list* in the parallel region.

If the number of threads requested exceeds the system resources available, the program stops.

The **omp_set_num_threads** function sets the first value of *num_list*. The **omp_get_max_threads** function returns the first value of *num_list*.

If you specify the number of threads for a given parallel region more than once with different settings, the compiler uses the following precedence order to determine which setting takes effect:

1. The number of threads set using the **num_threads** clause takes precedence over that set using the **omp_set_num_threads** function.
2. The number of threads set using the **omp_set_num_threads** function takes precedence over that set using the OMP_NUM_THREADS environment variable.

3. The number of threads set using the OMP_NUM_THREADS environment variable takes precedence over that set using the **PARTHDS** suboption of the XLSMPOPTS environment variable.

Example

```
export OMP_NUM_THREADS=3,4,5
export OMP_DYNAMIC=false

// omp_get_max_threads() returns 3

#pragma omp parallel
{
// Three threads running the parallel region
// omp_get_max_threads() returns 4

  #pragma omp parallel if(0)
  {
  // One thread running the parallel region
  // omp_get_max_threads() returns 5

    #pragma omp parallel
    {
    // Five threads running the parallel region
    // omp_get_max_threads() returns 5
    }
  }
}
```

OMP_PROC_BIND: The OMP_PROC_BIND environment variable controls whether OpenMP threads can be moved between processors. The syntax is as follows:

By default, the OMP_PROC_BIND environment variable is not set. If you set OMP_PROC_BIND to TRUE, the threads are bound to processors. If you set OMP_PROC_BIND to FALSE, the threads can be moved between processors.

If you do not set OMP_PROC_BIND, but set the suboptions of XLSMPOPTS (**startproc**/**stride** or **procs**), the threads are bound to processors according to the settings in the XLSMPOPTS environment variable.

If you set neither OMP_PROC_BIND nor the suboptions of XLSMPOPTS (**startproc**/**stride** or **procs**), the threads are not bound to processors.

If you do not set OMP_PROC_BIND and the XLSMPOPTS setting (**startproc**/**stride** or **procs**) is invalid, the threads are not bound to processors.

If you set OMP_PROC_BIND to TRUE and also set the suboptions of XLSMPOPTS (**startproc**/**stride** or **procs**), the threads are bound to processors according to the settings in the XLSMPOPTS environment variable.

Notes:

- If **procs** is set and the number of CPU IDs specified is smaller than the number of threads used by the program, the remaining threads are not bound to processors.

- If XLSMPOPTS=**startproc** is used, the value specified by **startproc** is smaller than the number of CPUs, and the value specified by **stride** causes a thread to bind to a CPU outside the range of available processors, some of the threads are bound and some are not.

If you set OMP_PROC_BIND to TRUE, but do not set the XLSMPOPTS suboption (**startproc**/**stride** or **procs**), the threads are bound to processors.

If you set OMP_PROC_BIND to TRUE and the XLSMPOPTS setting (**startproc**/**stride** or **procs**) is invalid, the threads are bound to processors.

If you set OMP_PROC_BIND to FALSE and also set the suboptions of XLSMPOPTS (**startproc**/**stride** or **procs**), the threads are not bound to processors.

If you set OMP_PROC_BIND to FALSE, but do not set the suboptions of XLSMPOPTS (**startproc**/**stride** or **procs**), the threads are not bound to processors.

If you set OMP_PROC_BIND to FALSE and the XLSMPOPTS setting (**startproc**/**stride** or **procs**) is invalid, the threads are not bound to processors.

The following table summarizes the previous thread binding rules:

Table 8. Thread binding rule summary

OMP_PROC_BIND settings	**XLSMPOPTS settings**	**Thread binding results**
OMP_PROC_BIND is not set	XLSMPOPTS is not set	Threads are not bound
OMP_PROC_BIND is not set	XLSMPOPTS is set (**startproc**/**stride** or **procs**)	Threads are bound according to the settings in XLSMPOPTS
OMP_PROC_BIND is not set	XLSMPOPTS setting is invalid	Threads are not bound
OMP_PROC_BIND=TRUE	XLSMPOPTS is not set	Threads are bound

Table 8. Thread binding rule summary (continued)

OMP_PROC_BIND settings	XLSMPOPTS settings	Thread binding results
OMP_PROC_BIND=TRUE	XLSMPOPTS is set (**startproc**/**stride** or **procs**)	Threads are bound according to the settings in XLSMPOPTS **Notes:** • If **procs** is set and the number of CPU IDs specified is smaller than the number of threads used by the program, the remaining threads are not bound. • If XLSMPOPTS=**startproc** is used, the value specified by **startproc** is smaller than the number of CPUs, and the value specified by **stride** causes a thread to bind to a CPU outside the range of available processors, some of the threads are bound and some are not.
OMP_PROC_BIND=TRUE	XLSMPOPTS setting is invalid	Threads are bound
OMP_PROC_BIND=FALSE	XLSMPOPTS is not set	Threads are not bound
OMP_PROC_BIND=FALSE	XLSMPOPTS is set (**startproc**/**stride** or **procs**)	Threads are not bound
OMP_PROC_BIND=FALSE	XLSMPOPTS setting is invalid	Threads are not bound

Restriction: On the BG/Q system, the OMP_PROC_BIND environment variable is always set to TRUE. If you set OMP_PROC_BIND to FALSE, the compiler ignores it.

Note: The OMP_PROC_BIND environment variable provides a portable way to control whether OpenMP threads can be migrated. The **startproc**/**stride** or **procs** suboption of the XLSMPOPTS environment variable, which is an IBM extension, provides a finer control to bind OpenMP threads to processors. If portability of your application is important, use only the OMP_PROC_BIND environment variable to control thread binding.

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

For example:

```
OMP_SCHEDULE="guided, 4"
```

Valid options for *algorithm* are:

- auto

- dynamic[, *n*]
- guided[, *n*]
- runtime
- static[, *n*]

If specifying a chunk size with *n*, the value of *n* must be a positive integer.

The default scheduling algorithm is **auto**.

Related reference:

"omp_set_schedule" on page 500

"omp_get_schedule" on page 499

OMP_STACKSIZE:

The OMP_STACKSIZE environment variable indicates the stack size of threads created by the OpenMP run time. OMP_STACKSIZE sets the value of the *stacksize-var* internal control variable. OMP_STACKSIZE does not control the stack size of the master thread. The syntax is as follows:

By default, the size value is represented in Kilobytes. You can also use the suffixes B, K, M, or G if you want to indicate the size in Bytes, Kilobytes, Megabytes, or Gigabytes respectively. White space is allowed between and around the size value and the suffix. For example, these two examples both indicate a stack size of 10 Megabytes.

```
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
```

If OMP_STACKSIZE is not set, the initial value of the *stacksize-var* internal control variable is set to the default value. For 64-bit mode, the default is up to the limit imposed by system resources. If the compiler cannot use the stack size specified or if OMP_STACKSIZE does not conform to the correct format, the compiler sets the environment variable to the default value. If the STACK suboption of the XLSMPOPTS environment variable and the OMP_STACKSIZE environment are specified, the OMP_STACKSIZE environment variable takes precedence.

OMP_THREAD_LIMIT:

The **OMP_THREAD_LIMIT** environment variable sets the number of OpenMP threads to use for the whole program. The syntax is as follows:

n The number of OpenMP threads to use for the whole program. It must be a positive scalar integer.

The value for **OMP_THREAD_LIMIT** is a positive integer.

If the **OMP_THREAD_LIMIT** environment variable is not set and the **OMP_NUM_THREADS** environment variable is set to a single value, the default value for **OMP_THREAD_LIMIT** is the value of **OMP_NUM_THREADS** or the number of available processors, whichever is greater.

If the **OMP_THREAD_LIMIT** environment variable is not set and the **OMP_NUM_THREADS** environment variable is set to a list, the default value for **OMP_THREAD_LIMIT** is the multiplication of all the numbers in the list or the number of available processors, whichever is greater.

If both the **OMP_THREAD_LIMIT** and **OMP_NUM_THREADS** environment variables are not set, the default value for **OMP_THREAD_LIMIT** is the number of available processors.

OMP_WAIT_POLICY:

The OMP_WAIT_POLICY environment variable gives hints to the compiler about the preferred behavior of waiting threads during program run time. The OMP_WAIT_POLICY environment variable sets the *wait-policy-var* internal control variable value.

The syntax is as follows:

The default value for OMP_WAIT_POLICY is PASSIVE.

Use ACTIVE if you want waiting threads to be mostly active. With ACTIVE, the thread consumes processor cycles while waiting, if possible.

Note: If the hardware threads are oversubscribed or multiple user threads are bound to the same hardware thread, OMP_WAIT_POLICY=ACTIVE might cause dead lock.

Use PASSIVE if you want waiting threads to be mostly passive. That is, the preference is for the thread to not consume processor cycles while waiting. For example, you prefer waiting threads to sleep or to yield the processor to other threads.

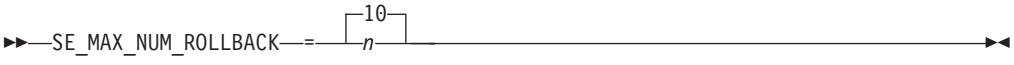
Note: If the OMP_WAIT_POLICY environment variable is set and the SPINS, YIELDS, or DELAYS suboptions of the XLSMPOPTS environment variable are specified, OMP_WAIT_POLICY takes precedence.

Environment variables for thread-level speculative execution

The environment variables for thread-level speculative execution have no effect unless the thread-level speculative execution is enabled with the "-qsmp=speculative" compiler option.

SE_MAX_NUM_ROLLBACK:

The SE_MAX_NUM_ROLLBACK environment variable indicates the maximum number a speculative thread of a particular thread-level speculative execution region can roll back before the region is run by one thread.



n The maximum number of times a thread for a speculative region can roll

back before being run by a single thread. It is a positive integer and must not exceed 2^{32}. The default value is 10.

SE_MAX_NUM_ROLLBACK = 0 forces the region of thread-level speculative execution to be run by one thread.

Related information

- Thread-level speculative execution

SE_REPORT_STAT_ENABLE:

The SE_REPORT_STAT_ENABLE environment variable enables or disables the statistics query function for thread-level speculative execution.

The default value is **NO**. The value is not case sensitive.

When SE_REPORT_STAT_ENABLE is set to YES, you can retrieve the statistics through the se_get_all_stats built-in function.

Related information

- Built-in functions for thread-level speculative execution

SE_REPORT_NAME:

The SE_REPORT_NAME environment variable specifies the name of the statistics log file and the location where it is created for thread-level speculative execution.

file_name
The name of the log file, or the directory and name of the log file. If you specify the name only, the file is placed in the current working directory.

If SE_REPORT_NAME is not set, the log file is placed in the current working directory and is named `se_report.log.`*rank* where *rank* is the MPI rank of the process that called the `se_print_stats` function.

Related information

- Built-in functions for thread-level speculative execution

SE_REPORT_LOG:

The SE_REPORT_LOG environment variable specifies how to create the statistics log files for thread-level speculative execution.

The value is not case sensitive.

SUMMARY
The statistics log file is generated only at the end of the program. It contains the statistics of all the regions of thread-level speculative execution for all hardware threads.

FUNC
The statistics log file is generated and updated at each call to the se_print_stats built-in function. It contains all the statistics at the time when the se_print_stats function is called.

ALL
The statistics log file is generated and updated at each call to the se_print_stats built-in function and at the end of the program.

VERBOSE
The statistics log file is generated and updated at each call to the se_print_stats built-in function and at the end of the program. The generated report file also includes the addresses of memory access conflicts during the speculation.

Related information

- Built-in functions for thread-level speculative execution

Environment variables for transactional memory

The environment variables for transactional memory have no effect unless transactional memory is enabled with the "-qtm" compiler option.

TM_MAX_NUM_ROLLBACK:

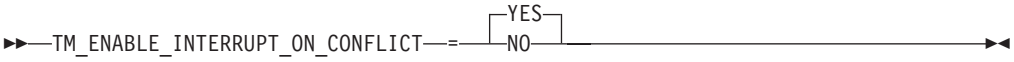
The TM_MAX_NUM_ROLLBACK environment variable specifies the maximum number a thread of a particular transactional atomic region can roll back before the thread goes into irrevocable mode.

n The maximum number of times a thread of a transactional atomic region can roll back before executing in irrevocable mode. It is a positive integer and must not exceed 2^{32}. The default value is 10.

TM_MAX_NUM_ROLLBACK = 0 forces the thread to enter irrevocable mode.

TM_ENABLE_INTERRUPT_ON_CONFLICT:

The TM_ENABLE_INTERRUPT_ON_CONFLICT environment variable indicates whether to generate interrupts on conflicts between speculative access.



The default value is **YES**. The value is not case sensitive.

Conflict resolution usually happens at the end of transactions. Use this environment variable to generate interrupts for access conflicts in transactional

atomic regions. This option is useful for long running transactions because the conflict resolution logic immediately starts inside the interrupt handler of the kernel.

Notes:

- Interrupts are always generated for speculative buffer overflows, supervised-mode violations, or conflicts between speculative and non-speculative access.
- Use with discretion if the TM_ENABLE_INTERRUPT_ON_CONFLICT environment variable is set to NO. If a transaction atomic region contains control flow that depends on some bad speculative data, the program may hang or the speculative thread may branch to invalid locations.

TM_REPORT_STAT_ENABLE:

The TM_REPORT_STAT_ENABLE environment variable enables or disables statistics query built-in functions for transactional memory.

The default value is **NO**. The value is not case sensitive.

When TM_REPORT_STAT_ENABLE is set to YES, you can retrieve the statistics through the following built-in functions:

- "tm_get_stats" on page 494
- "tm_get_all_stats" on page 495

Related information

- Built-in functions for transactional memory

TM_REPORT_NAME:

The TM_REPORT_NAME environment variable specifies the name of the statistics log file and the location where it is created for transactional memory.

file_name
The name of the log file, or the directory and name of the log file. If you specify the name only, the file is placed in the current working directory.

If TM_REPORT_NAME is not set, the log file is placed in the current working directory and is named `tm_report.log.`*rank* where *rank* in the extension is the MPI rank of the process that called the tm_print_stats.

Related information

- Built-in functions for transactional memory

TM_REPORT_LOG:

The TM_REPORT_LOG environment variable specifies how to create the statistics log file for transactional memory.

By default, no statistics log file is created.

The value is not case sensitive.

SUMMARY
The statistics log file is generated only at the end of the program.

FUNC
The statistics log file is generated and updated at each call to the tm_print_stats built-in function.

ALL
The statistics log file is generated and updated at each call to the tm_print_stats built-in function and at the end of the program.

VERBOSE
The statistics log file is generated and updated at each call to the tm_print_stats built-in function and at the end of the program. The generated report file contains the address of transactions that enter into irrevocable mode, and other useful information, such as the configuration of the runtime.

If TM_REPORT_LOG is set to either SUMMARY, ALL, or VERBOSE, a log file that contains the following statistics is also created when the program ends:

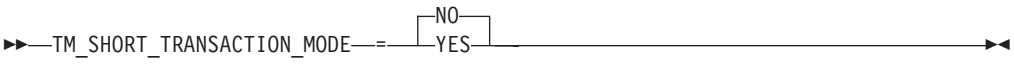
- It contains the cumulative statistics for transactional atomic regions that each hardware thread has run. The statistics follow the Structure of statistic counters.
- It contains a summary that shows the sum of all the statistic counters for all the hardware threads.

Related information

- Built-in functions for transactional memory

TM_SHORT_TRANSACTION_MODE:

The TM_SHORT_TRANSACTION_MODE environment variable indicates whether to use the short running speculation mode in the hardware.



The default value is **NO**. The value is not case sensitive.

TM_SHORT_TRANSACTION_MODE = YES enables the short running speculation mode. It is recommended if the transactional atomic region is short.

TM_SHORT_TRANSACTION_MODE = NO enables the long running speculation mode. It is recommended if the transactional atomic region is large and many reuses are among the references inside the transaction.

Using custom compiler configuration files

The XL C/C++ compiler generates a default configuration file /opt/ibmcmp/vac/bg/12.1/etc/vac.cfg.*$OSRelease.gcc$gccVersion*. For example, /opt/ibmcmp/vac/bg/12.1/etc/vac.cfg.sles11.gcc432 or /opt/ibmcmp/vac/bg/12.1/etc/vac.cfg.rhel6.2.gcc446 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 might want to leave the default configuration file as it is.

If you want users to be able to choose among several sets of compiler options, you might want to use custom configuration files for specific needs. For example, you might want to enable **-qlist** by default for compilations using the **bgxlc** compiler invocation command. This is to avoid forcing your users to specify this option on the command line for every compilation, because **-qlist** is automatically in effect every time the compiler is called with the **bgxlc** 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.

  Note: This option requires you to reapply your customization after you apply service to the compiler.
- 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 106
- "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 examines and processes 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 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, 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 which you see in the following example. 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, 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 attributes, such as **libraries** can also be used.

```
A: use =bgDEFLT
    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>
bgDEFLT:
    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 following example, the compiler references the **bgxlc** stanza in the user-defined configuration file and uses the option sets specified in the configuration files in the following order: *A1*, *A*, *D*, and *C*.

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

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

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

```
bgxlc:  use=bgDEFLT
    options=<A>

bgDEFLT:
    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 9. 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

```
bgxlc_prepend: use=bgxlc
            options-=<B1>
bgxlc_replace: use=bgxlc
            options:=<B2>
bgxlc_append: use=bgxlc
            options+=<B3>

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

Default configuration file vac.cfg

```
bgxlc: use=bgDEFLT
    options=<B>

bgDEFLT:
    options=<C>
```

The stanzas in the preceding configuration files use the following option sets, in the following orders:

1. stanza bgxlc uses *B*, *D*, and *C*
2. stanza bgxlc_prepend uses *B1*, *B*, *D*, and *C*
3. stanza bgxlc_replace uses *B2*
4. stanza bgxlc_append uses *B*, *D*, *C*, and *B3*

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

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

Figure 4. Using additional assignment operations

Examples of stanzas in custom configuration files

`bgDEFLT: use=bgDEFLT` `    options = -g`	This example specifies that the **-g** option is to be used in all compilations.

`bgxlc: use=bgxlc options+=-qlist`
`bgxlc_r: use=bgxlc_r`
`options+=-qlist`
	This example specifies that **-qlist** is to be used for any compilation called by the **bgxlc** and **bgxlc_r** commands. This **-qlist** specification overrides the default setting of **-qlist** specified in the system configuration file.

`bgDEFLT: use=bgDEFLT` `    libraries=-L/home/user/lib,-lmylib`	This example specifies that all compilations should link with /home/user/lib/libmylib.a.

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 5

Summary of compiler options by functional category

The XL C/C++ options available on the Blue Gene/Q platform 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 44
- "Language element control" on page 45
- "Template control (C++ only)" on page 46
- "Floating-point and integer control" on page 47
- "Error checking and debugging" on page 50
- "Listings, messages, and compiler information" on page 52
- "Optimization and tuning" on page 54
- "Object code control" on page 48
- "Linking" on page 57
- "Portability and migration" on page 58
- "Compiler customization" on page 58

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 10. Compiler output options

Option name	Equivalent pragma name	Description
"-c" on page 77	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 78	None.	When used in conjunction with the **-E** or **-P** options, preserves or removes comments in preprocessed output.
"-E" on page 100	None.	Preprocesses the source files named in the compiler invocation, without compiling, and writes the output to the standard output.

Table 10. Compiler output options (continued)

Option name	Equivalent pragma name	Description
"-qmakedep, -M" on page 198	None.	Creates an output file containing targets suitable for inclusion in a description file for the **make** command.
"-MF" on page 203	None.	Specifies the target for the output generated by the **-qmakedep** or **-M** options.
"-qmkshrobj" on page 205	None.	Creates a shared object from generated object files.
"-o" on page 206	None.	Specifies a name for the output object, assembler, or executable file.
"-P" on page 215	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 238	None.	Generates an assembler language file for each source file.
"-qshowmacros" on page 242	None.	Emits macro definitions to preprocessed output.
"-qtimestamps" on page 279	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 11. Compiler input options

Option name	Equivalent pragma name	Description
"-+ (plus sign) (C++ only)" on page 60	None.	Compiles any file as a C++ language file.
"-qcinc (C++ only)" on page 84	None.	Places an `extern "C" { }` wrapper around the contents of include files located in a specified directory.
"-I" on page 133	None.	Adds a directory to the search path for include files.
"-qidirfirst" on page 134	#pragma options idirfirst	Specifies whether the compiler searches for user include files in directories specified by the **-I** option *before* or *after* searching any other directories.
"-qinclude" on page 137	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 11. Compiler input options (continued)

Option name	Equivalent pragma name	Description
"-qsourcetype" on page 252	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 260	#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 12. Language element control options

Option name	Equivalent pragma name	Description
"-qasm" on page 71	None	Controls the interpretation and subsequent generation of code for assembler language extensions.
"-qcpluscmt (C only)" on page 89	None.	Enables recognition of C++-style comments in C source files.
"-D" on page 93	None.	Defines a macro as in a `#define` preprocessor directive.
"-qdigraph" on page 96	#pragma options digraph	Enables recognition of digraph key combinations or keywords to represent characters not found on some keyboards.
"-qdollar" on page 97	#pragma options dollar	Allows the dollar-sign ($) symbol to be used in the names of identifiers.
"-qignprag" on page 136	#pragma options ignprag	Instructs the compiler to ignore certain pragma statements.
"-qkeyword" on page 160	None.	Controls whether the specified name is treated as a keyword or as an identifier whenever it appears in your program source.
"-qlanglvl" on page 165	C #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 197	#pragma options long long	Allows IBM `long long` integer types in your program.

Table 12. Language element control options (continued)

Option name	Equivalent pragma name	Description
"-qmbcs, -qdbcs" on page 202	#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 256	None.	Controls whether inline functions are treated as having `static` or `extern` linkage.
"-qtabsize" on page 271	None.	Sets the default tab length, for the purposes of reporting the column number in error messages.
"-qtrigraph" on page 284	None.	Enables the recognition of trigraph key combinations to represent characters not found on some keyboards.
"-U" on page 285	None.	Undefines a macro defined by the compiler or by the **-D** compiler option.
"-qutf" on page 291	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 13. C++ template options

Option name	Equivalent pragma name	Description
"-qtempinc (C++ only)" on page 273	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 274	None.	Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.
"-qtemplaterecompile (C++ only)" on page 275	None.	Helps manage dependencies between compilation units that have been compiled using the **-qtemplateregistry** compiler option.

Table 13. C++ template options (continued)

Option name	Equivalent pragma name	Description
"-qtemplateregistry (C++ only)" on page 276	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 277	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 282	None.	Manages the implicit instantiation of templates.
"-qtmplparse (C++ only)" on page 283	None.	Controls whether parsing and semantic checking are applied to template definitions.
C++0x -qlanglvl=[no]externtemplate	None.	Suppresses the implicit instantiation of a template specialization or its members.
-qlanglvl=[no]gnu_externtemplate	None.	Suppresses the implicit instantiation of a template specialization or its members. This option is deprecated in XL C/C++ V12.1; you can use the option **-qlanglvl=[no]externtemplate** instead.
C++0x -qlanglvl=[no]variadic[templates]	None.	Defines class or function templates that can have any number (including zero) of parameters.

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 14. Floating-point and integer control options

Option name	Equivalent pragma name	Description
"-qbitfields" on page 76	None.	Specifies whether bit fields are signed or unsigned.
"-qchars" on page 81	#pragma options chars, #pragma chars	Determines whether all variables of type `char` are treated as either `signed` or `unsigned`.

Table 14. Floating-point and integer control options (continued)

Option name	Equivalent pragma name	Description
"-qenum" on page 102	#pragma options enum, #pragma enum	Specifies the amount of storage occupied by enumerations.
"-qfloat" on page 109	#pragma options float	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.
"-qldbl128" on page 186	#pragma options ldbl128	Increases the size of `long double` types from 64 bits to 128 bits.
"-qlonglit" on page 196	None.	In 64-bit mode, when determining the implicit types for integer literals, the compiler behaves as if an `l` or `L` suffix were added to integral literals with no suffix or with a suffix consisting only of `u` or `U`.
"-y" on page 301	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 15. Object code control options

Option name	Equivalent pragma name	Description
"-q64" on page 62	None.	Selects 64-bit compiler mode.
"-qalloca, -ma (C only)" on page 68	#pragma alloca	Provides an inline definition of system function `alloca` when it is called from source code that does not include the `alloca.h` header.
"-qcommon" on page 85	None.	Controls where uninitialized global variables are allocated.
"-qeh (C++ only)" on page 101	None.	Controls whether exception handling is enabled in the module being compiled.
"-qinlglue" on page 148	#pragma options inlglue	When used with **-O2** or higher optimization, inlines glue code that optimizes external function calls in your application.
"-qpic" on page 220	None.	Generates position-independent code suitable for use in shared libraries.

Table 15. Object code control options (continued)

Option name	Equivalent pragma name	Description
"-qppline" on page 222	None.	When used in conjunction with the **-E** or **-P** options, enables or disables the generation of `#line` directives.
"-qpriority (C++ only)" on page 224	#pragma options priority, #pragma priority	Specifies the priority level for the initialization of static objects.
"-qproto (C only)" on page 227	#pragma options proto	Specifies the linkage conventions for passing floating-point arguments to functions that have not been prototyped.
"-r" on page 228	None.	Produces a nonexecutable output file to use as an input file in another ld command call. This file may also contain unresolved symbols.
"-qreserved_reg" on page 232	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" on page 234	#pragma options ro, #pragma strings	Specifies the storage type for string literals.
"-qroconst" on page 235	#pragma options roconst	Specifies the storage location for constant values.
"-qrtti (C++ only)" on page 236	None.	Generates runtime type identification (RTTI) information for exception handling and for use by the `typeid` and `dynamic_cast` operators.
"-s" on page 237	None.	Strips the symbol table, line number information, and relocation information from the output file.
"-qsaveopt" on page 239	None.	Saves the command-line options used for compiling a source file, the user's configuration file name and the options specified in the configuration files, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.

Table 15. Object code control options (continued)

Option name	Equivalent pragma name	Description
"-qstackprotect" on page 255	None.	Provides protection against malicious code or programming errors that overwrite or corrupt the stack.
"-qstatsym" on page 259	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" on page 272	#pragma options tbtable	Controls the amount of debugging traceback information that is included in the object files.
"-qthreaded" on page 278	None.	Indicates to the compiler whether it must generate threadsafe code.
"-qtls" on page 279	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 299	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 52.

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

Table 16. Error checking and debugging options

Option name	Equivalent pragma name	Description
"-# (pound sign)" on page 61	None.	Previews the compilation steps specified on the command line, without actually invoking any compiler components.

Table 16. Error checking and debugging options (continued)

Option name	Equivalent pragma name	Description
"-qcheck" on page 83	#pragma options check	Generates code that performs certain types of runtime checking.
"-qflttrap" on page 113	#pragma options flttrap	Determines what types of floating-point exceptions to detect at run time.
"-qformat" on page 116	None.	Warns of possible problems with string input and output format specifications.
"-qfullpath" on page 118	#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.
"-qfunctrace" on page 118	None.	Calls the tracing routines to trace the entry and exit points of the specified functions in a compilation unit.
"-g" on page 121	None.	Generates debugging information for use by a symbolic debugger, and makes the program state available to the debugging session at selected source locations.
"-qhalt" on page 127	#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" on page 128	None.	Stops compilation before producing any object files, executable files, or assembler source files if a specified error message is generated.
"-qinfo" on page 139	#pragma options info, #pragma info	Produces or suppresses groups of informational messages.
"-qinitauto" on page 146	#pragma options initauto	Initializes uninitialized automatic variables to a specific value, for debugging purposes.
"-qkeepparm" on page 160	None.	When used with **-O2** or higher optimization, specifies whether procedure parameters are stored on the stack.
"-qlinedebug" on page 190	None.	Generates only line number and source file name information for a debugger.

Table 16. Error checking and debugging options (continued)

Option name	Equivalent pragma name	Description
"-qmaxerr" on page 199	None.	Stops compilation when the number of error messages of a specified severity level or higher reaches a specified number.
"-qoptdebug" on page 211	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 268	None.	Determines the information that appears in the symbol table.
"-qsyntaxonly (C only)" on page 269	None.	Performs syntax checking without generating an object file.
"-qwarn0x (C++0x)" on page 296	None.	Controls whether to inform users with messages about differences in their programs caused by migration from the C++98 standard to the C++0x standard.
"-qwarn64" on page 298	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 50 to provide a more robust overview of your application when checking for errors and unexpected behavior.

Table 17. Listings and messages options

Option name	Equivalent pragma name	Description
"-qattr" on page 74	#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 98	None.	Dumps a representation of the hierarchy and virtual function table layout of each class object to a file.
"-qflag" on page 107	#pragma options flag, C++ "#pragma report (C++ only)" on page 345	Limits the diagnostic messages to those of a specified severity level or higher.
"-qlist" on page 191	#pragma options list	Produces a compiler listing file that includes an object listing.

Table 17. Listings and messages options (continued)

Option name	Equivalent pragma name	Description
"-qlistfmt" on page 192	None.	Creates an XML or HTML report to assist with finding optimization opportunities.
"-qlistopt" on page 195	None.	Produces a compiler listing file that includes all options in effect at the time of compiler invocation.
"-qphsinfo" on page 219	None.	Reports the time taken in each compilation phase to standard output.
"-qprint" on page 223	None.	Enables or suppresses listings.
"-qreport" on page 230	None.	Produces listing files that show how sections of code have been optimized.
"-qshowinc" on page 241	#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.
"-qskipsrc" on page 245	None.	When a listing file is generated using the **-qsource** option, **-qskipsrc** can be used to determine whether the source statements skipped by the compiler are shown in the source section of the listing file. Alternatively, the **-qskipsrc=hide** option is used to hide the source statements skipped by the compiler.
"-qsource" on page 251	#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 254	None.	Adds the corresponding source code lines to diagnostic messages generated by the compiler.
"-qsuppress" on page 266	None.	Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.

Table 17. Listings and messages options (continued)

Option name	Equivalent pragma name	Description
"-v, -V" on page 291	None.	Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.
"-qversion" on page 292	None.	Displays the version and release of the compiler being invoked.
"-w" on page 294	None.	Suppresses informational, language-level and warning messages.
"-qxref" on page 299	#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 18. Optimization and tuning options

Option name	Equivalent pragma name	Description
"-qaggrcopy" on page 63	None.	Enables destructive copy operations for structures and unions.
"-qalias" on page 64	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 69	None.	Specifies the processor architecture for which the code (instructions) should be generated.

Table 18. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qcache" on page 79	None.	When specified with **-O4**, **-O5**, or **-qipa**, specifies the cache configuration for a specific execution machine.
"-qcompact" on page 86	#pragma options compact	Avoids optimizations that increase code size.
"-qdataimported, -qdatalocal, -qtocdata" on page 94	None.	Marks data as local or imported in 64-bit compilations.
"-qdirectstorage" on page 97	None.	Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.
"-qhot" on page 130	#pragma nosimd, #pragma novector	Performs high-order loop analysis and transformations (HOT) during optimization.
"-qignerrno" on page 135	#pragma options ignerrno	Allows the compiler to perform optimizations that assume `errno` is not modified by system calls.
"-qipa" on page 151	None.	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).
"-qisolated_call" on page 157	#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.
"-qlibansi" on page 188	#pragma options libansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
"-qlibmpi" on page 189	None.	Asserts that all functions with Message Passing Interface (MPI) names are in fact MPI functions and not a user function with different semantics.
"-qmaxmem" on page 201	#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" on page 204	None.	Controls the generation of the table of contents (TOC), which the compiler creates for an executable file in 64-bit compilation mode.

Table 18. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-O, -qoptimize" on page 207	#pragma options optimize	Specifies whether to optimize code during compilation and, if so, at which level.
"-p, -pg, -qprofile" on page 214	None.	Prepares the object files produced by the compiler for profiling.
"-qprefetch" on page 222	None.	Inserts prefetch instructions automatically where there are opportunities to improve code performance.
"-qprocimported, -qproclocal, -qprocunknown" on page 225	#pragma options procimported, #pragma options proclocal, #pragma options procunkown	Marks functions as local, imported, or unknown in 64-bit compilations.
"-qinline" on page 149	None.	Attempts to inline functions instead of generating calls to those functions, for improved performance.
"-qrestrict (C only)" on page 233	None.	Indicates to the compiler that no other pointers can access the same memory that has been addressed by function parameter pointers.
"-qsimd" on page 243	#pragma nosimd	Controls whether the compiler can automatically take advantage of vector instructions for processors that support them.
"-qsmallstack" on page 246	None.	Reduces the size of the stack frame.
"-qsmp" on page 247	None.	Enables parallelization of program code.
"-qstrict" on page 261	#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.
"-qstrict_induction" on page 265	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.
"-qtm" on page 281	None.	Enables transactional memory.

Table 18. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qtune" on page 284	#pragma options tune	Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.
"-qunroll" on page 286	#pragma options unroll, #pragma unroll	Controls loop unrolling, for improved performance.
"-qunwind" on page 289	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 19. Linking options

Option name	Equivalent pragma name	Description
"-qcrt" on page 90	None.	Specifies whether system startup files are to be linked.
"-e" on page 99	None.	When used together with the **-qmkshrobj**, specifies an entry point for a shared object.
"-L" on page 164	None.	At link time, searches the directory path for library files specified by the **-l** option.
"-l" on page 163	None.	Searches for the specified library file, lib*key*.so, and then lib*key*.a for dynamic linking, or just for lib*key*.a for static linking.
"-qlib" on page 187	None.	Specifies whether standard system libraries and XL C/C++ libraries are to be linked.
"-R" on page 229	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 257	None.	Controls how shared and nonshared runtime libraries are linked into an application.

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 20. Portability and migration options

Option name	Equivalent pragma name	Description
"-qabi_version (C++ only)" on page 63	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 66	#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 126	None.	Produces prototype declarations from K&R function definitions or function definitions with empty parentheses, and displays them to standard output.
"-qpack_semantic" on page 216	None.	Controls the syntax and semantics of the **#pragma pack** directive.
"-qupconv (C only)" on page 290	#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 alternative locations for compiler components, configuration files, standard include directories, and internal compiler operation. These options are useful for specialized installations, testing scenarios, and the specification of additional command-line options.

Table 21. Compiler customization options

Option name	Equivalent pragma name	Description
"-qasm_as" on page 72	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 75	None.	Determines substitute path names for XL C/C++ executables such as the compiler, assembler, linker, and preprocessor.
"-qcomplexgccincl" on page 87	#pragma complexgcc	Specifies whether to use GCC parameter-passing conventions for complex data types (equivalent to enabling **-qfloat=complexgcc**) for selected include files only.

Table 21. Compiler customization options (continued)

Option name	Equivalent pragma name	Description
"-qc_stdinc (C only)" on page 90	None.	Changes the standard search location for the XL C header files.
"-qcpp_stdinc (C++ only)" on page 91	None.	Changes the standard search location for the XL C++ header files.
"-F" on page 106	None.	Names an alternative configuration file or stanza for the compiler.
"-qgcc_c_stdinc (C only)" on page 124	None.	Changes the standard search location for the GNU C system header files.
"-qgcc_cpp_stdinc (C++ only)" on page 125	None.	Changes the standard search location for the GNU C++ system header files.
"-qpath" on page 217	None.	Determines substitute path names for XL C/C++ executables such as the compiler, assembler, and linker.
"-qoptfile" on page 212	None.	Specifies a file containing a list of additional command line options to be used for the compilation.
"-qspill" on page 254	#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 270	None.	Applies the prefix specified by the **-B** option to the designated components.
"-W" on page 295	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 303

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 381

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 **-qsourcetype=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.cplspls` as a C++ source file, enter:

```
bgxlc++ -+ myprogram.cplspls
```

Related information

- "-qsourcetype" on page 252

-# (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:

```
bgxlc myprogram.c -#
```

Related information

- "-v, -V" on page 291

-q64

Category

Object code control

Pragma equivalent

None.

Purpose

Selects 64-bit compiler mode.

Use the **-q64** option, along with the **-qarch** and **-qtune** compiler options, to optimize the output of the compiler to the architecture on which that output will be used.

Syntax

Defaults

-q64

Predefined macros

When **-q64** is in effect, __64BIT__ is defined to 1.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to run on a computer with a 64-bit Blue Gene® architecture, enter:

```
bgxlc -o testing myprogram.c -q64 -qarch=qp
```

Related information

- Specifying compiler options for architecture-specific compilation
- "-qarch" on page 69
- "-qtune" on page 284

-qabi_version (C++ only)

Category

Portability and migration

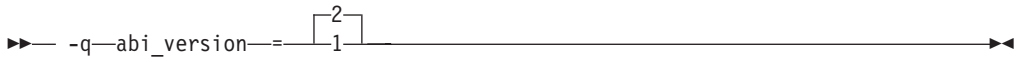
Pragma equivalent

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



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

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

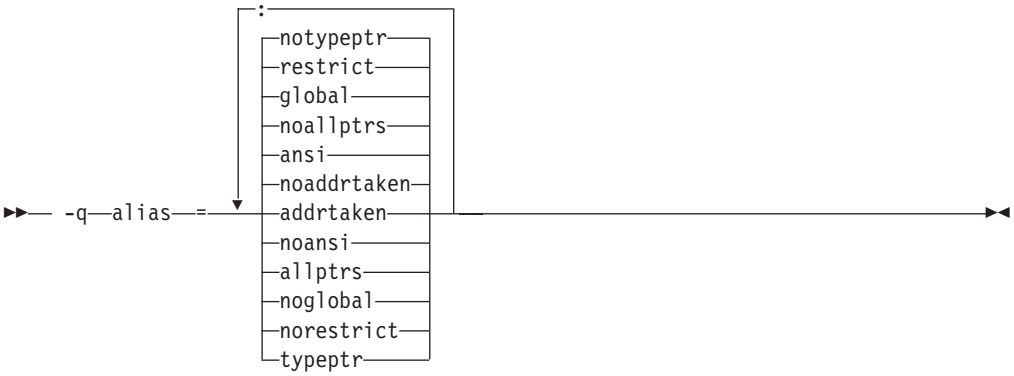
Pragma equivalent

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 **bgcc**.
  -qalias=noaddrtaken:noallptrs:noansi:global:restrict:notypeptr for the **bgcc** 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:

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

Related information

- "-qipa" on page 151
- -qinfo=als
- "#pragma disjoint" on page 315
- *Type-based aliasing* in the *XL C/C++ Language Reference*
- *The restrict type qualifier* in the *XL C/C++ Language Reference*
- "-qrestrict (C only)" on page 233

-qalign

Category

Portability and migration

Pragma equivalent

#pragma options align, #pragma align

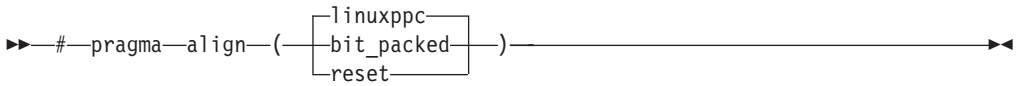
Purpose

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

Syntax

Option syntax

Pragma syntax



Defaults

-qalign=power

linuxppc

Parameters

bit_packed
Bit field data is packed on a bitwise basis without respect to byte boundaries.

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.

If the `vector4double` data is misaligned, the compiler issues a warning message.

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.

Note: When using **-qalign**, all system headers are also compiled with **-qalign**. 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 `bgxlc 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 `bgxlc 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 338
- "Aligning data" in the *XL C/C++ Optimization and Programming Guide*
- "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:

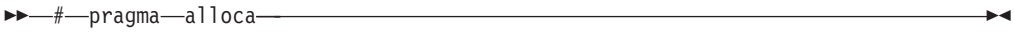
- You include the header file alloca.h
- You compile with **-Dalloca=__alloca**
- You directly call the built-in function using the form `__alloca`

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

Pragma syntax



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:

```
bgxlc myprogram.c -qalloca
```

Related information

- "-D" on page 93
- "__alignx" on page 485

-qarch

Category

Optimization and tuning

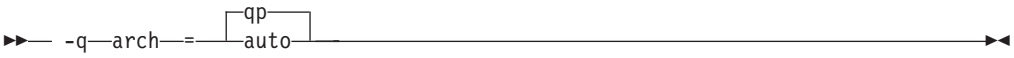
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

- **-qarch=qp**
- **-qarch=auto** when **-O4** or **-O5** is in effect.

Parameters

auto
When you compile your program with **bg**-prefixed invocation commands, **-qarch=auto** has the same effect as **-qarch=qp**.

qp Produces object code that runs on the Blue Gene/Q platforms. It has the following effects on other options or features:

- **-qarch=qp** also enables Blue Gene/Q vector data types.
- **-qarch=qp** automatically sets the **-qsimd=auto** option. You can disable SIMD optimization with the **-qsimd=noauto** option.
- When **-qhot** is set with **-qarch=qp**, **-qhot=fastmath:level=0** is also set by default.

Usage

For any given **-qarch** setting, the compiler defaults to a specific, matching **-qtune** setting, which can provide additional performance improvements. For detailed information on using **-qarch** and **-qtune** together, see "-qtune" on page 284.

For a given application program, make sure that you specify the same **-qarch** setting when you compile each of its source files.

Predefined macros

See "Macros related to architecture settings" on page 387 for a list of macros that are predefined by **-qarch** suboptions.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to run on a computer with the Blue Gene/Q architecture, enter:

```
bgxlc -o testing myprogram.c -qarch=qp
```

Related information

- -qprefetch
- -qfloat
- "-qtune" on page 284
- "Specifying compiler options for architecture-specific compilation" on page 9
- "-q64" on page 62
- "Macros related to architecture settings" on page 387
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-qasm

Category

Language element control

Pragma equivalent

None.

Purpose

Controls the interpretation 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”
- “-qlanglvl” on page 165
- “-qkeyword” on page 160
- "Inline assembly statements" in the *XL C/C++ Language Reference*
- "Keywords for language extensions"

-qasm_as

Category

Compiler customization

Pragma equivalent

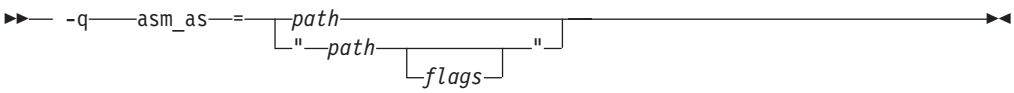
None.

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



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:

```
bgxlc 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:

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

Related information

- "-qasm" on page 71

-qassert

Category

Optimization and tuning

Pragma equivalent

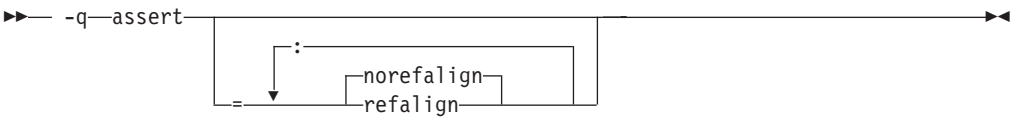
None

Purpose

Provides information about the characteristics of the files that can help to fine-tune optimizations.

Syntax

Option:



Defaults

-qassert=norefalign

Parameters

refalign | norefalign
Specifies that all pointers inside the compilation unit only point to data that is naturally aligned according to the length of the pointer types. With this assertion, the compiler might generate more efficient code. This assertion is particularly useful when you target a SIMD architecture with **-qhot=level=0** or **-qhot=level=1** with **-qsimd=auto**.

-qattr

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]attr

Purpose

Produces a compiler listing that includes the attribute component of the attribute and cross-reference section of the listing.

Syntax

Defaults

-qnoattr

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.

Note: Specifying **-qattr** does not list the `#define` directives. You can use "-qshowmacros" on page 242 instead.

Predefined macros

None.

Examples

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

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

Related information

- "-qshowmacros" on page 242
- "-qprint" on page 223
- "-qxref" on page 299

-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

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 270 for a list of these. If you use the **-B** option without **-t***programs*, 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:

```
bgxlc -tcbI -B/home/jim/opt/ibmcmp/vacpp/bg/12.1/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 217
- "-t" on page 270
- "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

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

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:

```
bgxlc myprogram.c -c
```

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

```
bgxlc myprogram.c -c -o new.o
```

Related information

- "-E" on page 100
- "-o" on page 206
- "-P" on page 215
- "-qsyntaxonly (C only)" on page 269

-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

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:

```
bgxlc myprogram.c -P -C
```

Related information

- "-E" on page 100
- "-P" on page 215

-qcache

Category

Optimization and tuning

Pragma equivalent

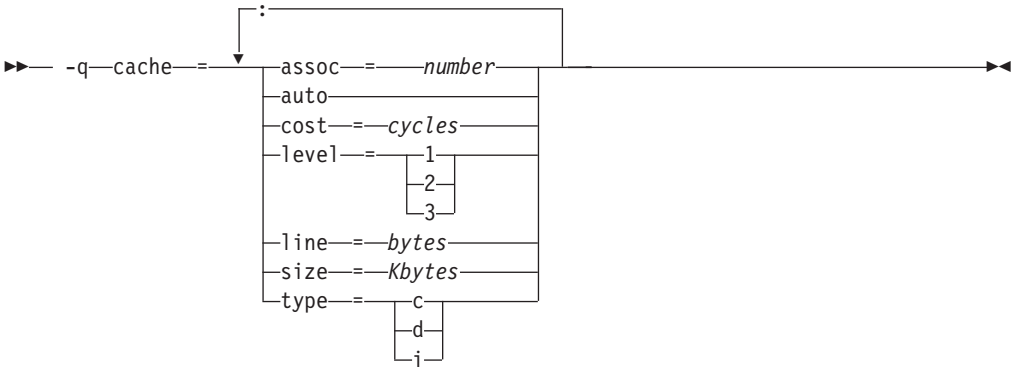
None.

Purpose

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

If you know the type of execution system for a program, and that system has its instruction or data cache configured differently from the default case, use this option to specify the exact cache characteristics. The compiler uses this information to calculate the benefits of cache-related optimizations.

Syntax



Defaults

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

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

auto
Automatically detects the specific cache configuration of the compiling machine. This assumes that the execution environment will be the same as the compilation environment.

cost
Specifies the performance penalty resulting from a cache miss.

cycles

level
Specifies the level of cache affected. If a machine has more than one level of cache, use a separate **-qcache** option.

level
Is one of:

1 Basic cache

2 Level-2 cache or, if there is no level-2 cache, the table lookaside buffer (TLB)

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

Usage

The **-qtune** setting determines the optimal default **-qcache** settings for most typical compilations. You can use the **-qcache** to override these default settings. However,

if you specify the wrong values for the cache configuration, or run the program on a machine with a different configuration, the program will work correctly but may be slightly slower.

You must specify **-O4**, **-O5**, or **-qipa** with the **-qcache** option.

Use the following guidelines when specifying **-qcache** suboptions:

- Specify information for as many configuration parameters as possible.
- If the target execution system has more than one level of cache, use a separate **-qcache** option to describe each cache level.
- If you are unsure of the exact size of the cache(s) on the target execution machine, specify an estimated cache size on the small side. It is better to leave some cache memory unused than it is to experience cache misses or page faults from specifying a cache size larger than actually present.
- The data cache has a greater effect on program performance than the instruction cache. If you have limited time available to experiment with different cache configurations, determine the optimal configuration specifications for the data cache first.
- If you specify the wrong values for the cache configuration, or run the program on a machine with a different configuration, program performance may degrade but program output will still be as expected.
- The **-O4** and **-O5** optimization options automatically select the cache characteristics of the compiling machine. If you specify the **-qcache** option together with the **-O4** or **-O5** options, the option specified last takes precedence.

Predefined macros

None.

Examples

To tune performance for a system with a combined instruction and data level-1 cache, where cache is 2-way associative, 8 KB in size and has 64-byte cache lines, enter:

```
bgxlc -O4 -qcache=type=c:level=1:size=8:line=64:assoc=2 file.c
```

Related information

- “-qcache” on page 79
- “-O, -qoptimize” on page 207
- “-qtune” on page 284
- “-qipa” on page 151
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-qchars

Category

Floating-point and integer control

Pragma equivalent

#pragma options chars, #pragma chars

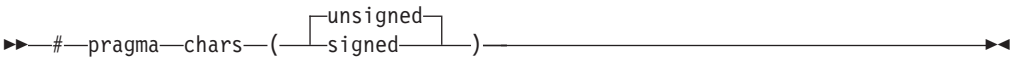
Purpose

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

Syntax

Option syntax

Pragma syntax



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:

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

-qcheck

Category

Error checking and debugging

Pragma equivalent

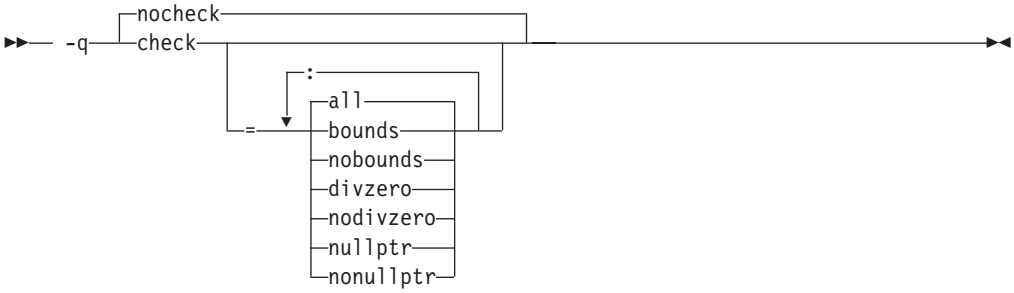
#pragma options [no]check

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. Note that the runtime checks may result in slower application execution.

Syntax



Defaults

-qnocheck

Parameters

all
Enables all suboptions.

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.

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:

```
bgxlc 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

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:

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

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

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

-qcommon

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

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

- C++ **-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" on page 205
- "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

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:

```
bgxlc myprogram.c -O -qcompact
```

-qcomplexgccincl

Category

Compiler customization

Pragma equivalent

None.

Purpose

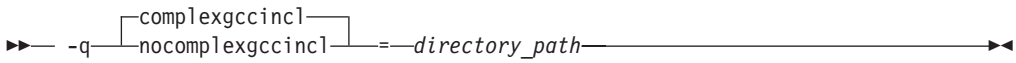
Specifies whether to use GCC parameter-passing conventions for complex data types (equivalent to enabling **-qfloat=complexgcc**) for selected include files only.

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.

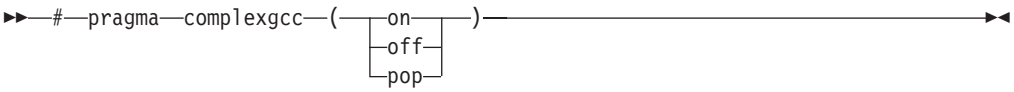
You can also use the pragma directives to enable or disable GCC parameter-passing conventions for complex data types for selected files or sections of code.

Syntax

Option syntax



Pragma syntax



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 11.

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=nogccomplex** for the code that follows it. This instructs the compiler to use Blue Gene/Q 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 109

-qcpluscmt (C only)
Category

Language element control

Pragma equivalent

None.

Purpose

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

Syntax

Defaults
- **-qcpluscmt** when the **bgxlc** or **bgc99** 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:

```
bgxlc 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 78
- "-qlanglvl" on page 165
- "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

Defaults

-qcrt

Predefined macros

None.

Related information

- "-qlib" on page 187

-qc_stdinc (C only)

Category

Compiler customization

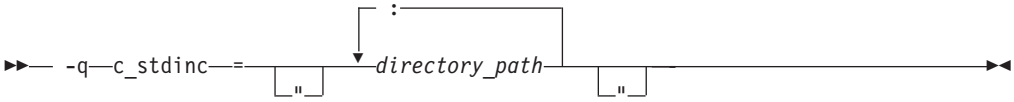
Pragma equivalent

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/vacpp/bg/12.1/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 11 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:

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

Related information

- "-qgcc_c_stdinc (C only)" on page 124
- "-qstdinc" on page 260
- "-qinclude" on page 137
- "Directory search sequence for include files" on page 11
- "Specifying compiler options in a configuration file" on page 7

-qcpp_stdinc (C++ only)

Category

Compiler customization

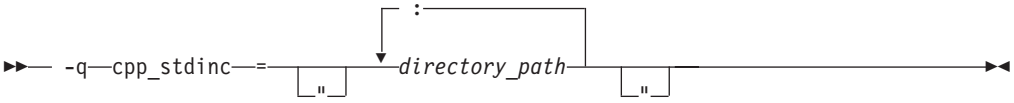
Pragma equivalent

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/vacpp/bg/12.1/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 11 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:

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

Related information

- "-qgcc_cpp_stdinc (C++ only)" on page 125
- "-qstdinc" on page 260
- "-qinclude" on page 137

- "Directory search sequence for include files" on page 11
- "Specifying compiler options in a configuration file" on page 7

-D

Category

Language element control

Pragma equivalent

None.

Purpose

Defines a macro as in a `#define` preprocessor directive.

Syntax

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 **-U***name* option, which is used to undefine macros defined by the **-D** option, has a higher precedence than the **-D***name* 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:

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

Related information

- "-U" on page 285
- Chapter 5, "Compiler predefined macros," on page 381

-qdataimported, -qdatalocal, -qtocdata

Category

Optimization and tuning

Pragma equivalent

None.

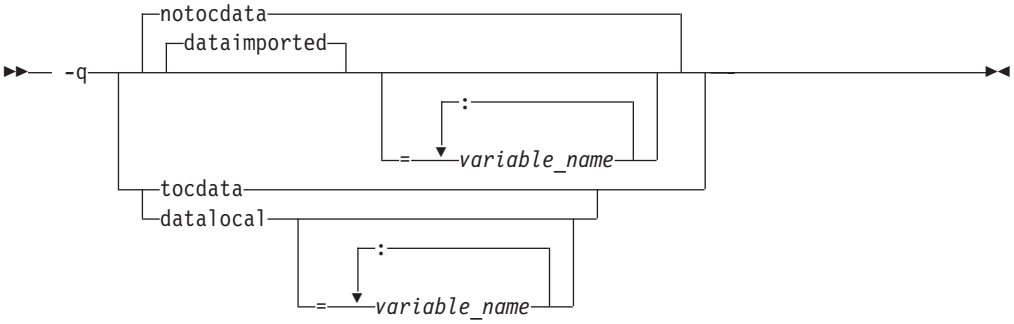
Purpose

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

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, -qproclocal, -qprocunknown" on page 225

-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 **-qsymtab=unref** option.

Syntax

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:

```
bgxlc myprogram.c -g -qdbxextra
```

Related information

- "-qfullpath" on page 118
- "-qlinedebug" on page 190
- "-g" on page 121
- "#pragma options" on page 334
- "-qsymtab (C only)" on page 268

-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

Defaults

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

```
bgxlc myprogram.c -qnodigraph
```

Related information

- "-qlanglvl" on page 165
- "-qtrigraph" on page 284

-qdirectstorage

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

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

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:

```
bgxlc myprogram.c -qdollar
```

Related information

- "-qlanglvl" on page 165

-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

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:

```
bgxlc++ 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

Defaults

None.

Parameters

name
The name of the entry point for the shared executable.

Usage

Specify the **-e** option only with the **-qmkshrobj** option. For more information, see the description for **-qmkshrobj**.

Note: When you link object files, do not use the **-e** option. The default entry point of the executable output is `__start`. Changing this label with the **-e** flag can produce errors.

Predefined macros

None.

Related information

- "-qmkshrobj" on page 205

-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

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:

```
bgxlc 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

- "-qppline" on page 222
- "-C, -C!" on page 78
- "-P" on page 215
- "-qsyntaxonly (C only)" on page 269

-qeh (C++ 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

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)" on page 236

-qenum

Category

Floating-point and integer control

Pragma equivalent

#pragma options enum, #pragma enum

Purpose

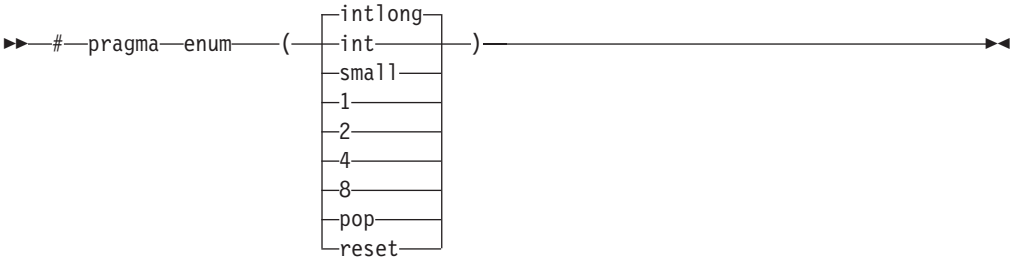
Specifies the amount of storage occupied by enumerations.

C++0x The **-qenum** option affects only unscoped enumerations that have no fixed underlying type. For enumerations with a fixed underlying type, the **-qenum** option is ignored. C++0x

Syntax

Option syntax

Pragma syntax



Defaults

-qenum=intlong

Parameters

1 Specifies that enumerations occupy 1 byte of storage, are of type `signed 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. C Values cannot exceed the range of `signed int`.

4 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 64-bit compilation mode, the enumeration is of type `long` if the range of enumeration values falls within the limits of `signed long`, and `unsigned long` otherwise.

int

C Specifies that enumerations occupy 4 bytes of storage and are of type `int`.

C++ 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.

intlong

Specifies that enumerations occupy 8 bytes of storage, as with the **8** suboption, if the range of values in the enumeration cannot be represented by one of `int` or `unsigned int`. Otherwise, the enumerations occupy 4 bytes of storage as with the **4** suboption.

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. Signedness 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 22. Enumeration sizes and types

	enum=1	enum=2	enum=4	enum=8 (64-bit compilation mode)

Table 22. Enumeration sizes and types (continued)

Range	var	const	var	const	var	const	var	const
0..127	signed char	int	short	int	int	int	long	long
-128..127	signed char	int	short	int	int	int	long	long
0..255	unsigned char	int	short	int	int	int	long	long
0..32767	ERROR[1]	int	short	int	int	int	long	long
-32768..32767	ERROR[1]	int	short	int	int	int	long	long
0..65535	ERROR[1]	int	unsigned short	int	int	int	long	long
0..2147483647	ERROR[1]	int	ERROR[1]	int	int	int	long	long
-(2147483647+1) ..2147483647	ERROR[1]	int	ERROR[1]	int	int	int	long	long
0..4294967295	ERROR[1]	unsigned int[2]	ERROR[1]	unsigned int[2]	unsigned int[2]	unsigned int[2]	long	long
0..(2^{63}-1)	ERROR[1]	long[2]	ERROR[1]	long[2]	ERROR[1]	long[2]	long[2]	long[2]
-2^{63}..(2^{63}-1)	ERROR[1]	long[2]	ERROR[1]	long[2]	ERROR[1]	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[2]	unsigned long[2]

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

Notes:

- These enumerations are too large for the **-qenum=1|2|4|** C **int** C setting. 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**.
- C Enumeration types must not exceed the range of `int` when compiling C applications to ISO C 1989 and ISO C 1999 Standards. With the **stdc89 | stdc99** language level in effect, the compiler will behave as follows if the value of an enumeration exceeds the range of `int` and the **-qenum** option in effect supports this value:
  - If **-qenum=int** is in effect, a severe error message is issued and compilation stops.

  - For all other settings of **-qenum**, an informational message is issued and compilation continues.
- When `-qenum=8`, for all ranges of enumerator values up to 2^{32}-1, the table identifies the underlying type to be `long` for 64-bit compilation mode. This violates the rule in the standard: The underlying type should not be larger than `int` if enumerator values fit in `int` or `unsigned int`.
- When `-qenum=small`, for enumerators in the range of 0-2147483647 with at least one enumerator having a value bigger than `65535`, the table identifies the underlying type to be `unsigned int`, which cannot be promoted to `int`. This violates the rule in the standard: Promotion could be to `int`, because it can hold the enumeration values in the range of 0-2147483647.

The **#pragma enum** directive must 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
#pragma enum=reset /* second reset isn't required */
```

Predefined macros

None.

-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 **bgxlc**, but you specify the **bgc99** stanza, the compiler will use all the settings specified in the **bgc99** 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:

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

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

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

To compile `myprogram.c` using the stanza `bgc99` you have created in a configuration file called `/usr/tmp/myconfig.cfg`, enter:

```
bgxlc myprogram.c -F/usr/tmp/myconfig.cfg:bgxlf95bgc99
```

Related information

- "Using custom compiler configuration files" on page 39
- "-B" on page 75
- "-t" on page 270
- "-W" on page 295
- "Specifying compiler options in a configuration file" on page 7
- "Compile-time and link-time environment variables" on page 22

-qflag

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options flag, "#pragma report (C++ only)" on page 345

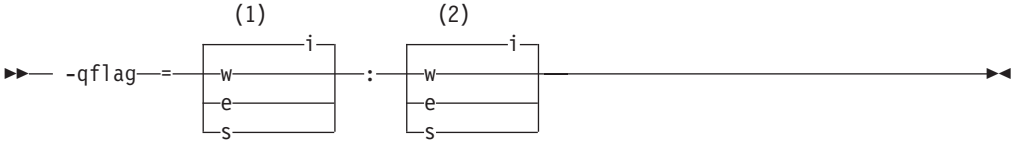
Purpose

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

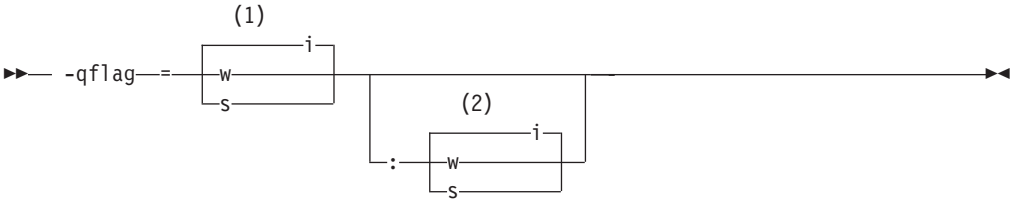


Notes:

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.

C **e**
Specifies that only error (E), severe error (S), and unrecoverable error (U) messages are to display.

s C Specifies that only severe error (S) and unrecoverable error (U) messages are to display. C++ Specifies that only severe error (S) messages are to display.

Usage

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

C++ 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.

The **-qhaltonmsg** option has precedence over the **-qflag** option. If both **-qhaltonmsg** and **-qflag** are specified, messages that are not selected by **-qflag** are also printed.

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:

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

Related information

- "-qinfo" on page 139
- "-qhaltonmsg" on page 128
- "-w" on page 294
- "Compiler messages" on page 14

-qfloat

Category

Floating-point and integer control

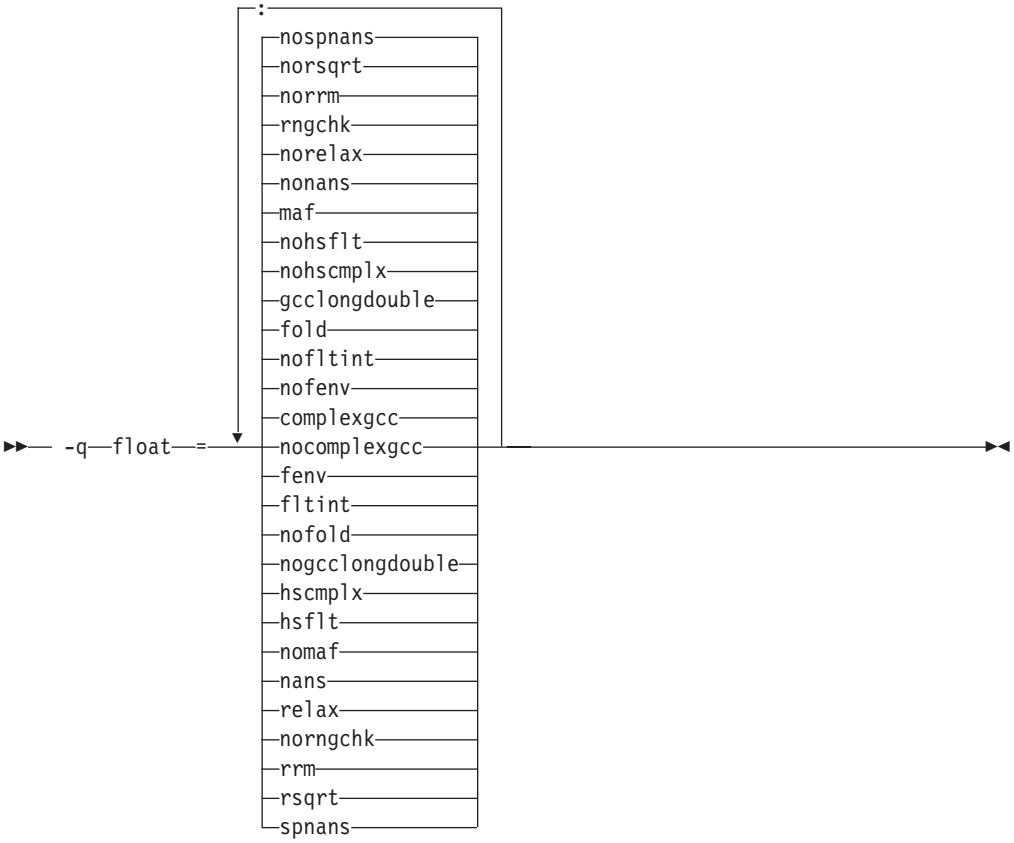
Pragma equivalent

#pragma options float

Purpose

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

Syntax



Defaults

- **-qfloat=nocomplexgcc**
- **-qfloat=complexgcc:nofenv:nofltint:fold:gcclongdouble:nohscmplx:nohsflt:maf:nonans:norelax:rngchk:norrm:norsqrt:nospnans**
- **-qfloat=fltint:rsqrt:norngchk** when **-qnostrict**, **-qstrict=nooperationprecision:noexceptions**, or **-O3** or higher optimization level is in effect.

Parameters

complexgcc | nocomplexgcc
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 165 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
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
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 "Binary floating-point built-in functions" on page 400 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
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=noieeefp** 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
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**
Generates extra instructions to detect signalling NaN on conversion from single-precision to double-precision.

Note:

- For details about the relationship between **-qfloat** suboptions and their **-qstrict** counterparts, see "-qstrict" on page 261.

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

Examples

To compile myprogram.c so that constant floating-point expressions are evaluated at compile time and multiply-add instructions are not generated, enter:

```
bgxlc myprogram.c -qfloat=fold:nomaf
```

Related information

- "-qarch" on page 69
- "-qcomplexgccincl" on page 87
- "-qflttrap"
- "-qldbl128" on page 186
- "-qstrict" on page 261

-qflttrap

Category

Error checking and debugging

Pragma equivalent

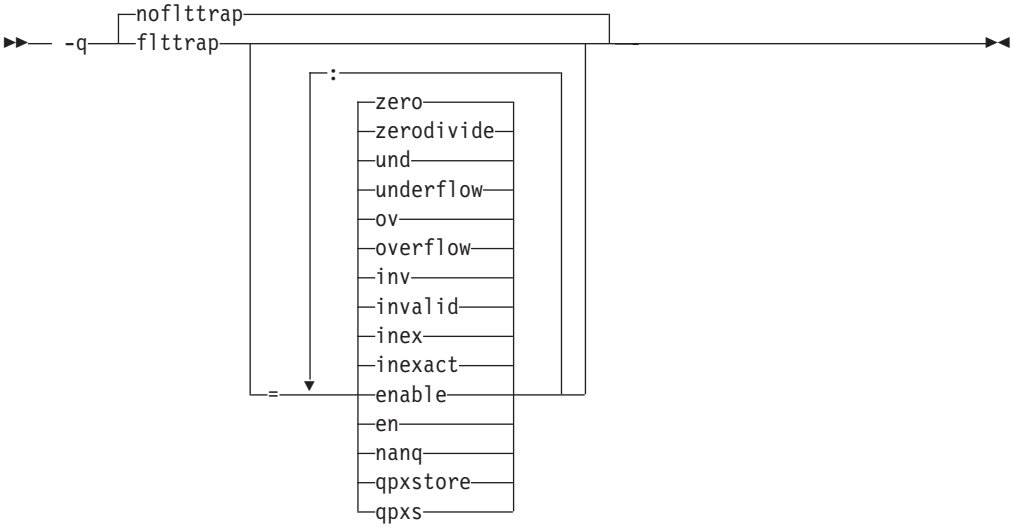
#pragma options [no]flttrap

Purpose

Determines what types of floating-point exceptions to detect at run time.

The program receives a **SIGFPE** signal when the corresponding exception occurs.

Syntax



Defaults

-qnoflttrap

Parameters

enable, en
Inserts a trap 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 Not a Number Quiet (NaNQ) and Not a Number Signalling (NaNS) 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.

qpxstore, qpxs
Enables the detection of Not a Number (NaN) or infinity values in Quad Processing eXtension (QPX) vectors. The exceptions only occur on QPX store instructions.

To detect NaN or infinity values, the compiler generates stores with indicating instructions for QPX vectors in registers. The indicating vector stores are used for both stores as a result of using QPX store intrinsics or assignment operators.

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.

Usage

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.

The **-qflttrap=qpxstore** suboption detects exceptions only on QPX store instructions, while other suboptions detect exceptions on scalar floating point operations.

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.

The **-qflttrap** 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.

Note: Due to the transformations performed and the exception handling support of some vector instructions, use of **-qsimd=auto** may change the location where an exception is caught or even cause the compiler to miss catching an exception.

Predefined macros

None.

Example

```
#include <stdio.h>

int main()
{
  float x, y, z;
  x = 5.0;
  y = 0.0;
  z = x / y;
  printf("%f", z);
}
```

When you compile this program with the following command, the program stops when the division is performed.

```
bgxlc -qfloat=zerodivide:enable divide_by_zero.c
```

The **zerodivide** suboption identifies the type of exception to guard against. The **enable** suboption causes a **SIGFPE** signal to be generated when the exception occurs.

Related information

- "-qfloat" on page 109
- "-qarch" on page 69

-qformat

Category

Error checking and debugging

Pragma equivalent

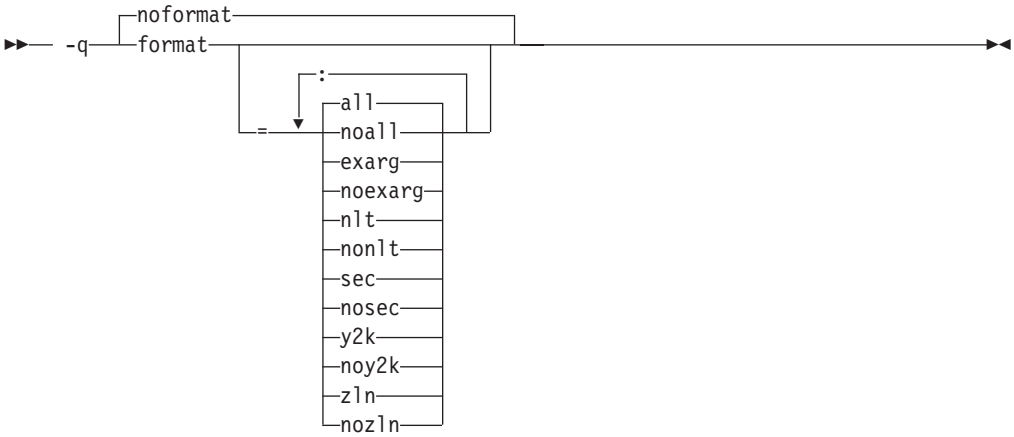
None.

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:

```
bgxlc myprogram.c -qformat=all
bgxlc myprogram.c -qformat
```

To enable all format diagnostic checking except that for y2k date diagnostics, enter:

```
bgxlc 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

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 190
- "-g" on page 121

-qfunctrace

Category

Error checking and debugging

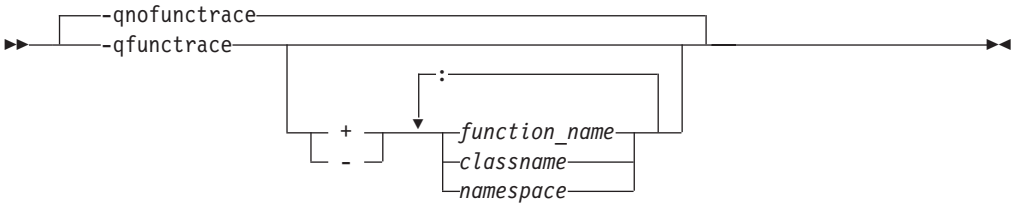
Pragma equivalent

None.

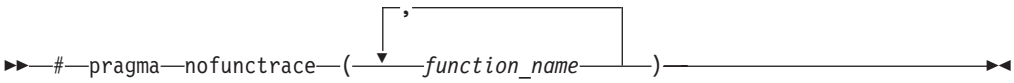
Purpose

Calls the tracing routines to trace the entry and exit points of the specified functions in a compilation unit.

Syntax



Pragma syntax



Defaults

-qnofunctrace

Note: When **-qfunctrace** is specified for a C++ program, the functions in the *std* namespace are not traced by default.

Parameters

+ Instructs the compiler to trace *function_name*, *classes*, or *namespace*, and all its internal functions.

- Instructs the compiler not to trace *function_name*, *classes*, or *namespace*, or any of its internal functions.

function_name
Indicates the named functions to be traced.

classname
Indicates the named class to be traced.

namespace
Indicates the namespace to be traced.

Usage

-qfunctrace enables tracing for all functions in your program. **-qnofunctrace** disables tracing that was enabled by **-qfunctrace**.

The **-qfunctrace+** and **-qfunctrace-** suboptions enable tracing for a specific list of functions and are not affected by **-qnofunctrace**. The list of functions is cumulative.

This option inserts calls to the tracing functions that you have defined. These functions must be provided at the link step. For details about the interface of tracing functions, as well as when they are called, see the Tracing functions in your code section in the *XL C/C++ Optimization and Programming Guide*.

Use **+** or **-** to indicate the function , classname, or namespace to be traced by the compiler. For example, if you want to trace function `x`, use `-qfunctrace+x`. To trace a list of functions, you must use a colon `:` to separate them.

Two colons in a row `::` is a scope qualifier, you can use it to indicate C++ qualified names. For example, use `-qfunctrace+A::B:C` traces functions that begin with qualifiers `A::B` or `C`.

If you want to trace functions in your code, you can write tracing functions in your code by using the following C function prototypes:

- Use `void __func_trace_enter(const char *const function_name, const char *const file_name, int line_number, void **const user_data);` to define the entry point tracing routine.
- Use `void __func_trace_exit(const char *const function_name, const char *const file_name, int line_number, void **const user_data);` to define the exit point tracing routine.
- Use `void __func_trace_catch(const char *const function_name, const char *const file_name, int line_number, void **const user_data);` to define the catch tracing routine.

You must define your functions when you write the preceding function prototypes in your code.

For details about the these function prototypes as well as when they are called, see the **Tracing functions in your code** section in the *XL C/C++ Optimization and Programming Guide*.

Note:

- You can only use **+** and **-** one at a time. Do not use both of them together in the same **-qfunctrace** invocation.
- Definition of an inline function is traced. It is only the calls that have been inlined are not traced.

Predefined macros

None.

Examples

To trace functions `x`, `y`, and `z`, use `-qfunctrace+x:y:z`.

To trace all functions except for `x`, use `-qfunctrace -qfunctrace-x`.

The **-qfunctrace+** and **-qfunctrace-** suboptions only enable or disable tracing on the given list of cumulative functions. When functions, classes, and namespaces are used, the most completely specified option is in effect. The following is a list of examples:

- `-qfunctrace+x -qfunctrace+y` or `-qfunctrace+x -qnofunctrace -qfunctrace+y` enables tracing for only `x` and `y`.
- `-qfunctrace-x -qfunctrace` or `-qfunctrace -qfunctrace-x` traces all functions in the compilation unit except for `x`.
- `-qfunctrace -qfunctrace+x` traces all functions.
- `-qfunctrace+y -qnofunctrace` traces `y` only.
- If `functionX` is a member function of `classX`, then `-qfunctrace-classX::functionX -qfunctrace+classX` or `-qfunctrace+classX`

`-qfunctrace-classX::functionX` traces all member functions of `classX` but not `functionX`. This is because `classX::functionX` is more completely specified than `classX`. The more completely specified option has precedence over the less completely specified option.
- `-qfunctrace+MyClass` traces all member functions in `MyClass`.
- `-qfunctrace+std::vector` traces all instantiations of `std::vector`.
- `-qfunctrace+ABC -qfunctrace-ABC::foo` traces all functions defined in namespace `ABC` except for `foo`.

Related information
- For details about **#pragma nofunctrace**, see "#pragma nofunctrace" on page 333.
- For detailed information about how to implement function tracing routines in your code, as well as detailed examples and a list of rules for using them, see **Tracing functions in your code** in the *XL C/C++ Optimization and Programming Guide*.

-g

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates debugging information for use by a symbolic debugger, and makes the program state available to the debugging session at selected source locations.

Program state refers to the values of user variables at certain points during the execution of a program.

You can use different **-g** levels to balance between debug capability and compiler optimization. Higher **-g** levels provide a more complete debug support, at the cost of runtime or possible compile-time performance, while lower **-g** levels provide higher runtime performance, at the cost of some capability in the debugging session.

When the **-O2** optimization level is in effect, the debug capability is completely supported.

Note: When an optimization level higher than **-O2** is in effect, the debug capability is limited.

Syntax

Defaults

-g0

Parameters

-g

- When no optimization is enabled (**-qnoopt**), **-g** is equivalent to **-g9**.
- When the **-O2** optimization level is in effect, **-g** is equivalent to **-g2**.

-g0 Generates no debugging information. No program state is preserved.

-g1 Generates minimal read-only debugging information about line numbers and source file names. No program state is preserved.

-g2 Generates read-only debugging information about line numbers, source file names, and variables.

When the **-O2** optimization level is in effect, no program state is preserved.

-g3, -g4

Generates read-only debugging information about line numbers, source file names, and variables.

When the **-O2** optimization level is in effect:

- No program state is preserved.
- Function parameter values are available to the debugger at the beginning of each function.

-g5, -g6, -g7

Generates read-only debugging information about line numbers, source file names, and variables.

When the **-O2** optimization level is in effect:

- Program state is available to the debugger at `if` constructs, loop constructs, function definitions, and function calls. For details, see "Usage" on page 123.
- Function parameter values are available to the debugger at the beginning of each function.

-g8 Generates read-only debugging information about line numbers, source file names, and variables.

When the **-O2** optimization level is in effect:

- Program state is available to the debugger at the beginning of every executable statement.

- Function parameter values are available to the debugger at the beginning of each function.

-g9 Generates debugging information about line numbers, source file names, and variables. You can modify the value of the variables in the debugger.

When the **-O2** optimization level is in effect:

- Program state is available to the debugger at the beginning of every executable statement.
- Function parameter values are available to the debugger at the beginning of each function.

Usage

When no optimization is enabled, the debugging information is always available if you specify **-g2** or a higher level. When the **-O2** optimization level is in effect, the debugging information is available at selected source locations if you specify **-g5** or a higher level.

When you specify **-g8** or **-g9** with **-O2**, the debugging information is available at every source line with an executable statement.

When you specify **-g5**, **-g6**, or **-g7** with **-O2**, the debugging information is available for the following language constructs:

- `if` constructs

  The debugging information is available at the beginning of every `if` statement, namely at the line where the `if` keyword is specified. It is also available at the beginning of the next executable statement right after the `if` construct.
- Loop constructs

  The debugging information is available at the beginning of every `do`, `for`, or `while` statement, namely at the line where the `do`, `for`, or `while` keyword is specified. It is also available at the beginning of the next executable statement right after the `do`, `for`, or `while` construct.
- Function definitions

  The debugging information is available at the first executable statement in the body of the function.
- Function calls

  The debugging information is available at the beginning of every statement where a user-defined function is called. It is also available at the beginning of the next executable statement right after the statement that contains the function call.

Examples

Use the following command to compile `myprogram.c` and generate an executable program called `testing` for debugging:

```
bgxlc myprogram.c -o testing -g
```

The following command uses a specific **-g** level with **-O2** to compile `myprogram.c` and generate debugging information:

```
bgxlc myprogram.c -O2 -g8
```

Related information

- "#pragma ibm snapshot" on page 325
- "-qlinedebug" on page 190
- "-qfullpath" on page 118
- "-O, -qoptimize" on page 207
- "-qkeepparm" on page 160

-qgcc_c_stdinc (C only)

Category

Compiler customization

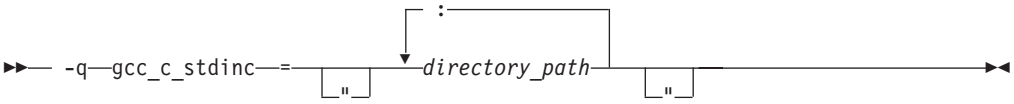
Pragma equivalent

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 11 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:

```
bgxlc myprogram.c -qgcc_c_stdinc=mypath/headers1:mypath:headers2
```

Related information

- "-qc_stdinc (C only)" on page 90
- "-qstdinc" on page 260
- "-qinclude" on page 137
- "Directory search sequence for include files" on page 11
- "Specifying compiler options in a configuration file" on page 7

-qgcc_cpp_stdinc (C++ only)

Category

Compiler customization

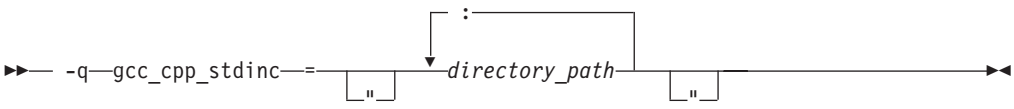
Pragma equivalent

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 11 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:

```
bgxlc++ myprogram.C -qgcc_cpp_stdinc=mypath/headers1:mypath:headers2
```

Related information

- "-qcpp_stdinc (C++ only)" on page 91
- "-qstdinc" on page 260
- "-qinclude" on page 137
- "Directory search sequence for include files" on page 11
- "Specifying compiler options in a configuration file" on page 7

-qgenproto (C only)

Category

Portability and migration

Pragma equivalent

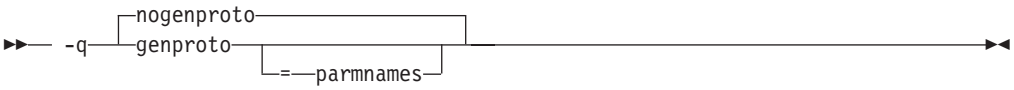
None.

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 16.

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 **-qhaltonmsg** 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:

```
bgxlc myprogram.c -qhalt=w
```

Related information

- "-qhaltonmsg"
- "-qflag" on page 107
- "-qmaxerr" on page 199

-qhaltonmsg

Category

Error checking and debugging

Pragma equivalent

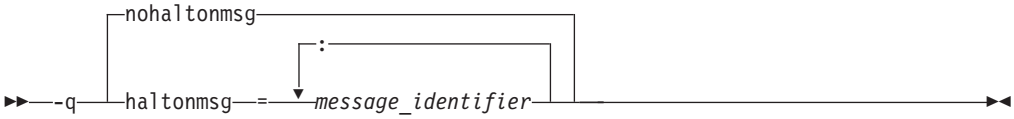
None.

Purpose

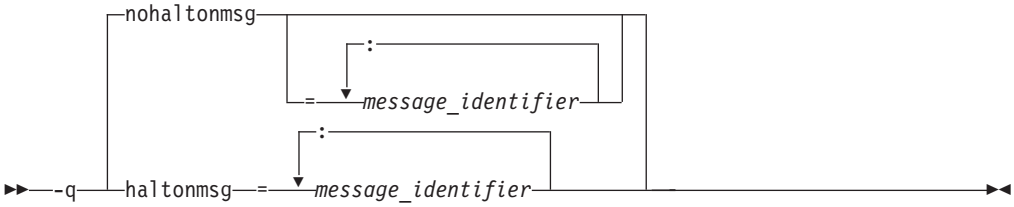
Stops compilation before producing any object files, executable files, or assembler source files if a specified error message is generated.

Syntax

-qhaltonmsg syntax - C



-qhaltonmsg syntax - C++



Defaults

-qnohaltonmsg

Parameters

message_identifier
Represents a message identifier. The message identifier must be in the following format:

15*dd*-*number*

where:

15 Is the compiler product identifier.

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 codes.

number
Is the message number.

Usage

When the compiler stops as a result of the **-qhaltonmsg** option, the compiler return code is nonzero. The severity level of a message that is specified by **-qhaltonmsg** is changed to S if its original severity level is lower than S.

You cannot specify **-qnohaltonmsg** to resume compilation if a message whose severity level is S has been issued.

C The **-qnohaltonmsg** compiler option cancels previous settings of **-qhaltonmsg**. C

C++ When you specify **-qnohaltonmsg** with message identifiers, the previous **-qhaltonmsg** instances with the same message identifiers lose effect. When you specify **-qnohaltonmsg** without specific message identifiers, all previous **-qhaltonmsg** instances lose effect.

If you specify two or three of the following options, the last option specified has precedence:

-qhaltonmsg=*message_identifier*

-qnohaltonmsg=*message_identifier*

-qnohaltonmsg

C++

-qhaltonmsg has precedence over **-qsuppress** and **-qflag**.

Predefined macros

None.

Related information

- "-qflag" on page 107
- "-qhalt" on page 127
- "Compiler messages" on page 14
- "-qsuppress" on page 266

-qhot

Category

Optimization and tuning

Pragma equivalent

#pragma novector

Purpose

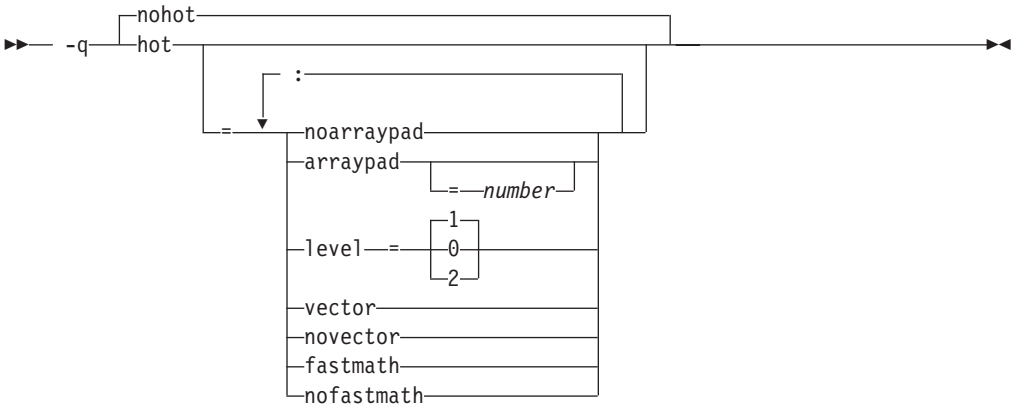
Performs high-order loop analysis and transformations (HOT) during optimization.

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:novector:fastmath** when **-O3** is in effect.
- **-qhot=noarraypad:level=1:vector:fastmath** when **-qsmp**, **-O4** or **-O5** is in effect.
- Specifying **-qhot** without suboptions is equivalent to **-qhot=noarraypad:level=1:vector:fastmath**.

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:noarraypad:fastmath**.

level=1 (option only)
Performs the default set of high-order transformations.

level=2 (option only)
Performs the default set of high-order transformations and some more aggressive loop transformations. **-qhot=level=2** must be used with **-qsmp**. This option performs aggressive loop analysis and transformations to improve cache reuse and exploit loop parallelization opportunities.

vector (option only) | novector
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. 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.

fastmath | nofastmath
You can use this suboption to tune your application to either use fast scalar versions of math functions or use the default versions.

For C/C++, you must use this suboption together with **-qignerrno**, unless **-qignerrno** is already enabled by other options.

-qhot=fastmath enables the replacement of math routines with available math routines from the XLOPT library only if **-qstrict=nolibrary** is enabled. **-qhot=nofastmath** disables the replacement of math routines by the XLOPT library. **-qhot=fastmath** is enabled by default if **-qhot** is specified regardless of the hot level.

Usage

If you do not also specify an optimization level when specifying **-qhot** on the command line, the compiler assumes **-O2**.

If you want to override the default **level** setting of **1** when using **-qsmp**, **-O4** or **-O5**, be sure to specify **-qhot=level=0** or **-qhot=level=2** *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 use the **-qreport** option in conjunction with **-qhot** or any optimization option that implies **-qhot** to produce a pseudo-C report showing how the loops were transformed. The loop transformations are included in the listing report if either the option **-qreport** or **-qlistfmt** is also specified. This `LOOP TRANSFORMATION SECTION` of the listing file also contains information about data prefetch insertion locations. For more information, see "-qreport" on page 230.

Predefined macros

None.

Related information

- "-qarch" on page 69
- "-qsimd" on page 243
- "-qprefetch" on page 222
- "-qreport" on page 230
- "-O, -qoptimize" on page 207
- "-qstrict" on page 261
- "-qsmp" on page 247
- *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

Defaults

See "Directory search sequence for include files" on page 11 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:

```
bgxlc myprogram.c -I/usr/tmp -I/oldstuff/history
```

Related information

- "-qidirfirst"
- "-qstdinc" on page 260
- "-qinclude" on page 137
- "Directory search sequence for include files" on page 11
- "Specifying compiler options in a configuration file" on page 7

-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

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 11.) 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:

```
bgxlc myprogram.c -I/usr/tmp/myinclude -qidirfirst
```

Related information

- "-I" on page 133
- "-qinclude" on page 137
- "-qstdinc" on page 260
- "-qc_stdinc (C only)" on page 90
- "-qcpp_stdinc (C++ only)" on page 91
- "Directory search sequence for include files" on page 11

-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

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 207

-qignprag

Category

Language element control

Pragma equivalent

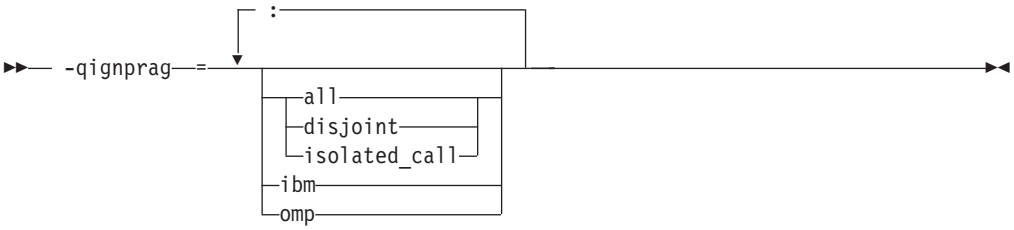
#pragma options [no]ignprag

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

C Ignores all **#pragma ibm snapshot** directives in the source file.

isolated_call

Ignores all **#pragma isolated_call** directives in the source file.

omp

Ignores all OpenMP parallel processing directives in the source file, such as **#pragma omp parallel**, **#pragma omp critical**.

Predefined macros

None.

Examples

To compile `myprogram.c` and ignore any **#pragma isolated_call** directives, enter:

```
bgxlc myprogram.c -qignprag=isolated_call
```

Related information

- "#pragma disjoint" on page 315
- "-qisolated_call" on page 157
- "#pragma ibm snapshot" on page 325
- "Pragma directives for parallel processing" on page 355

-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

Defaults

-qnoinclude

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 11.

Usage

The usage of the **-qinclude** option is similar to that of the `#include` directive. This section describes the differences between using **-qinclude** and `#include`.

The **-qinclude** option applies only to the files specified in the same compilation 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**, or 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.

Specifying **-qnoinclude** ignores any previous specification of `-qinclude`. Only the specifications of **-qinclude** after**-qnoinclude** are effective.

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.

The following rules apply when you use **-qinclude** with other options:

- 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.
- After searching the directory from which the compiler was invoked, **-qinclude** searches additional search paths added to the search chain by the **-I** option. You can specify the **-I** option before or after the **-qinclude** option.
- Files specified with **-qinclude** are included as dependencies in the **-M** output. However, the paths are different in the dependency file for the **-qinclude** option and the `#include` directive, because the files specified with the **-qinclude** option are searched in the invocation path first, whereas files included by the `#include` directive are not.

  When a dependency file is created as a result of a first build with the **-qinclude** option, a subsequent build without the **-qinclude** option will trigger recompile if the header file on the **-qinclude** option was touched between the two builds.
- In the compiler listing file generated by the **-qlistopt** option, each use of the **-qinclude** option has a separate entry in the `OPTION SECTION`.
- If both the **-qsource** option and the **-qinclude** option are used, header files specified with **-qinclude** are not included in the program source listing as `#include` directives. However, the files specified on `#include` directives in source programs are included.

Predefined macros

None.

Examples

To include the files `test1.h` and `test2.h` in the source file `test.c`, enter the following command:

```
bgxlc -qinclude=test1.h test.c  -qinclude=test2.h
```

Related information

- "Directory search sequence for include files" on page 11

-qinfo

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]info, #pragma info

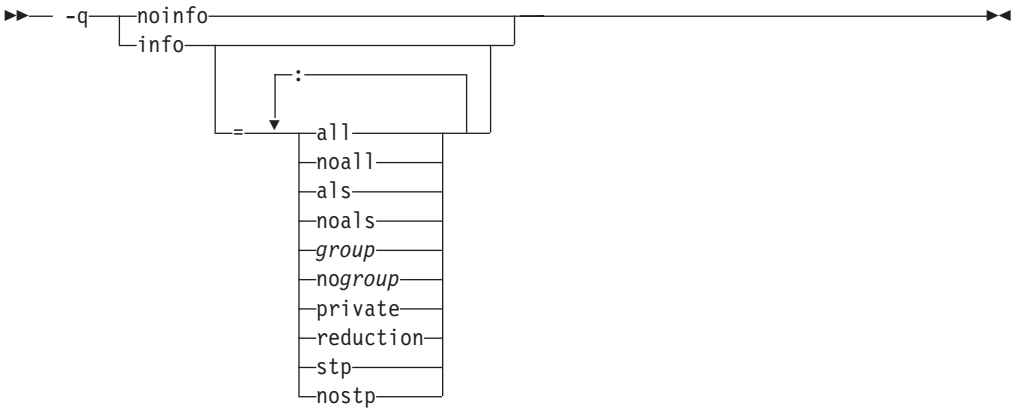
Purpose

Produces or suppresses groups of informational messages.

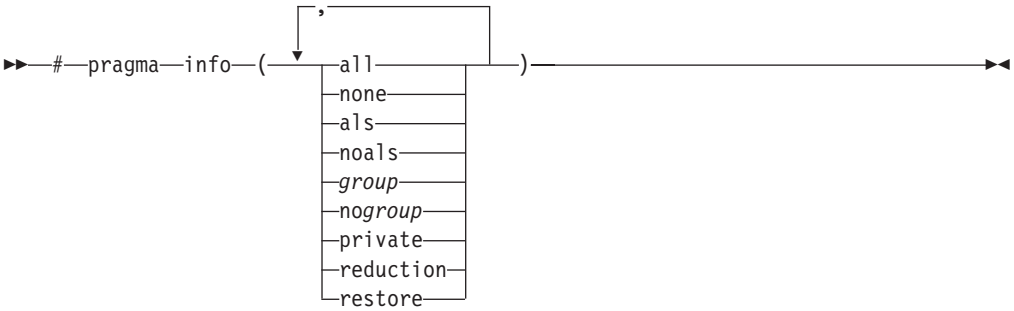
The messages are written to standard output and, optionally, to the listing file if one is generated.

Syntax

Option syntax



Pragma syntax



Defaults

-qnoinfo

- C -qnoinfo
- C++ -qinfo=lan:trx

Parameters

all
Enables all diagnostic messages for all groups except **als** and **ppt**.

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.

group | **no***group*
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.

C++ **vft | novft**
Generation of virtual function tables.

private
This suboption is deprecated and replaced by "-qreport" on page 230.

reduction
This suboption is deprecated and replaced by "-qreport" on page 230.

stp | nostp
Issues warnings for procedures that are not protected against stack corruption. **-qinfo=stp** has no effects unless the **-qstackprotect** option is also enabled. Like other **-qinfo** options, **-qinfo=stp** is enabled or disabled through **-qinfo=all / noall**. **-qinfo=nostp** is the default option.

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.

Usage

Specifying **-qinfo** with no suboptions is equivalent to **-qinfo=all**.

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 unreached statements, enter:

```
bgxlc 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;
}
bgxlC -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;
}

bgxlC -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;
}

bgxlC -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 107
- "-qreport" on page 230

-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

Defaults

-qnoinitauto

Parameters

hex_value
A one- to eight-digit hexadecimal number.

- To initialize each byte of storage to a specific value, specify one or two digits for the *hex_value*.
- To initialize each word of storage to a specific value, specify three to eight digits for the *hex_value*.
- In the case where less than the maximum number of digits are specified for the size of the initializer requested, leading zeros are assumed.
- In the case of word initialization, if an automatic variable is smaller than a multiple of 4 bytes in length, the *hex_value* is truncated on the left to fit. For example, if an automatic variable is only 1 byte and you specify five digits for the *hex_value*, the compiler truncates the three digits on the left and assigns the other two digits on the right to the variable. See Example 1.
- If an automatic variable is larger than the *hex_value* in length, the compiler repeats the *hex_value* and assigns it to the variable. See Example 1.
- If the automatic variable is an array, the *hex_value* is copied into the memory location of the array in a repeating pattern, beginning at the first memory location of the array. See Example 2.
- You can specify alphabetic digits as either uppercase or lowercase.
- The *hex_value* can be optionally prefixed with `0x`, in which `x` is case-insensitive.

Usage

The -qinitauto option provides the following benefits:

- Setting *hex_value* to zero ensures that all automatic variables are cleared before being used.
- You can use this option to initialize variables of real or complex type to a signaling or quiet NaN, which helps locate uninitialized variables in your program.

This option generates extra code to initialize the value of automatic variables. It reduces the runtime performance of the program and is to be used for debugging purposes only.

Restriction: Objects that are equivalenced, structure components, and array elements are not initialized individually. Instead, the entire storage sequence is initialized collectively.

Predefined macros

- __INITAUTO__ is defined to the *hex_value* that is specified on the **-qinitauto** option or pragma; otherwise, it is undefined.
- __INITAUTO_W__ is defined to the *hex_value*, repeated 4 times, specified on the **-qinitauto** option or pragma; otherwise, it is undefined.

Examples

Example 1: Use the **-qinitauto** option to initialize automatic variables of scalar types.

```
#include <stdio.h>

int main()
{
  char a;
  short b;
  int c;
  long long int d;

  printf("char a = 0x%X\n",(char)a);
  printf("short b = 0x%X\n",(short)b);
  printf("int c = 0x%X\n",c);
  printf("long long int d = 0x%X\n",d);
}
```

If you compile the program with `-qinitauto=AABBCCDD`, for example, the result is as follows:

```
char a = 0xDD
short b = 0xFFFFCCDD
int c = 0xAABBCCDD
long long int d = 0xAABBCCDDAABBCCDD
```

Example 2: Use the **-qinitauto** option to initialize automatic array variables.

```
#include <stdio.h>
#define ARRAY_SIZE 5

int main()
{
  char a[5];
  short b[5];
  int c[5];
  long long int d[5];

  printf("array of char: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
```

```
    printf("0x%1X ",(unsigned)a[i]);
  printf("\n");

  printf("array of short: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)b[i]);
  printf("\n");

  printf("array of int: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)c[i]);
  printf("\n");

  printf("array of long long int: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)d[i]);
  printf("\n");
}
```

If you compile the program with `-qinitauto=AABBCCDD`, for example, the result is as follows:

```
array of char: 0xAA 0xBB 0xCC 0xDD 0xAA
array of short: 0xAABB 0xCCDD 0xAABB 0xCCDD 0xAABB
array of int: 0xAABBCCDD 0xAABBCCDD 0xAABBCCDD 0xAABBCCDD 0xAABBCCDD
array of long long int: 0xAABBCCDDAABBCCDD 0xAABBCCDDAABBCCDD 0xAABBCCDDAABBCCDD
0xAABBCCDDAABBCCDD 0xAABBCCDDAABBCCDD
```

-qinlglue

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

Defaults

- **-qnoinlglue**
- **-qinlglue** when **-qtune=auto** is in effect.

Usage

Inlining glue code can cause the code size to grow. Specifying **-qcompact** overrides the **-qinlglue** setting to prevent code growth. If you want **-qinlglue** to be enabled, do not specify **-qcompact**.

Specifying **-qnoinlglue** or **-qcompact** can degrade performance; use these options with discretion.

The **-qinlglue** 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 86
- "-qprocimported, -qproclocal, -qprocunknown" on page 225
- "-qtune" on page 284

-qinline

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Attempts to inline functions instead of generating calls to those functions, for improved performance.

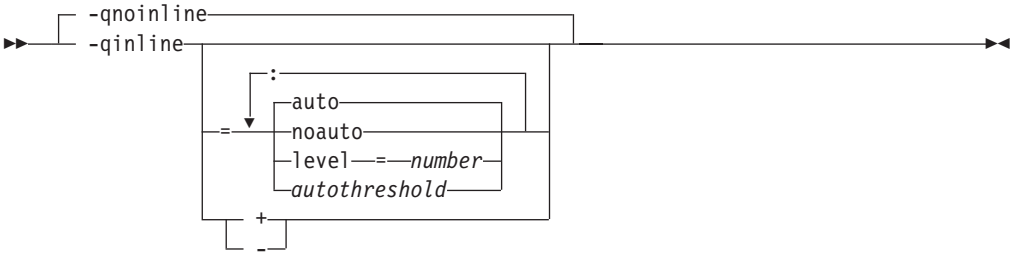
C++ Specifying **-qinline** enables automatic inlining by the compiler front end. Specifying **-qinline** with **-O** provides additional inlining by enabling inlining by the low-level optimizer. In both cases, the compiler attempts to inline all functions, in addition to those defined inside a class declaration or explicitly marked with the `inline` specifier.

C You must specify a minimum optimization level of **-O2** along with **-qinline** to enable inlining of functions, including those declared with the `inline` specifier. You can also use the **-qinline** option to specify restrictions on the functions that should or should not be inlined.

In all cases where **-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 **-qnoinline** disables all inlining, including that performed by the high-level optimizer with the **-qipa** option, and functions declared explicitly as `inline`.

Syntax



Defaults

- **-qnoinline**
- At optimization levels of **-O2** and higher, the default is **-qinline=noauto**
- **-qinline=auto:level=5** is the default suboption of **-qinline**

Parameters

noauto | **auto**
Enables or disables automatic inlining. If you do not specify any **-qinline** suboptions, **-qinline=auto** is the default.

Note: At optimization levels of **-O2** and higher, the default is **-qinline=auto**

level=*number*
Provides guidance to the compiler about the relative value of inlining. The values you specify for *number* must be positive integers between 0 and 10 inclusive. The default value for *number* is 5. If you specify a value less than 5, it implies less inlining. A value greater than 5 implies more inlining than the default.

C *autothreshold*
Represents the number of executable statements in a function. The number of executable statements in a function must be fewer than or equal to *autothreshold* for it to be considered for inlining. The value you specify for *autothreshold* must be a positive integer. The default value for *autothreshold* is 20. If you specify a value of 0, no functions are inlined. As you can see in the following example:

```
increment()
{
  int a, b, i;
   for (i=0; i<10; i++) /* statement 1 */
   {
      a=i;              /* statement 2 */
      b=i;              /* statement 3 */
   }
}
```

Usage

To maximize inlining, specify optimization (**-O**) and also specify the appropriate **-qinline** 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 debugging information, inlining may be suppressed.

Predefined macros

None.

Examples

To compile `myprogram.c` so that no functions are inlined, enter:

```
bgxlc myprogram.c -O2 -qnoinline
```

Assuming you have functions `salary`, `taxes`, `expenses`, and `benefits`, to compile `myprogram.c` so that the compiler tries to inline these functions, you enter:

```
bgxlc myprogram.c -O2 -qinline+salary:taxes:expenses:benefits
```

If you do not want the functions `salary`, `taxes`, `expenses`, and `benefits` to be inlined when you compile `myprogram.cmyprogram.f`, you enter:

```
bgxlc myprogram.c -O2 -qinline-salary:taxes:expenses:benefits
```

C In general, you can turn off automatic inlining and request that specific functions be inlined by naming them with the **+** form:

```
-O2 -qinline=noauto -qinline+salary:taxes:benefits
```

This causes only the functions named `salary`, `taxes`, or `benefits` to be inlined, if possible, and no others.

If you want to use the automatic inlining function, you use the `auto` suboption:

```
-O2 -qinline=auto
```

You can specify an inlining level between 6 and 10 to perform more aggressive automatic inlining. For example:

```
-O2 -qinline=auto:level=7
```

If automatic inlining is already enabled by default and you want to specify an inlining level (For example: 7), you enter:

```
-O2 -qinline=level=7
```

Related information

- “-g” on page 121
- “-qipa”
- “-O, -qoptimize” on page 207
- "The inline function specifier" in the *XL C/C++ Language Reference*

-qipa

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables or customizes a class of optimizations known as interprocedural analysis (IPA).

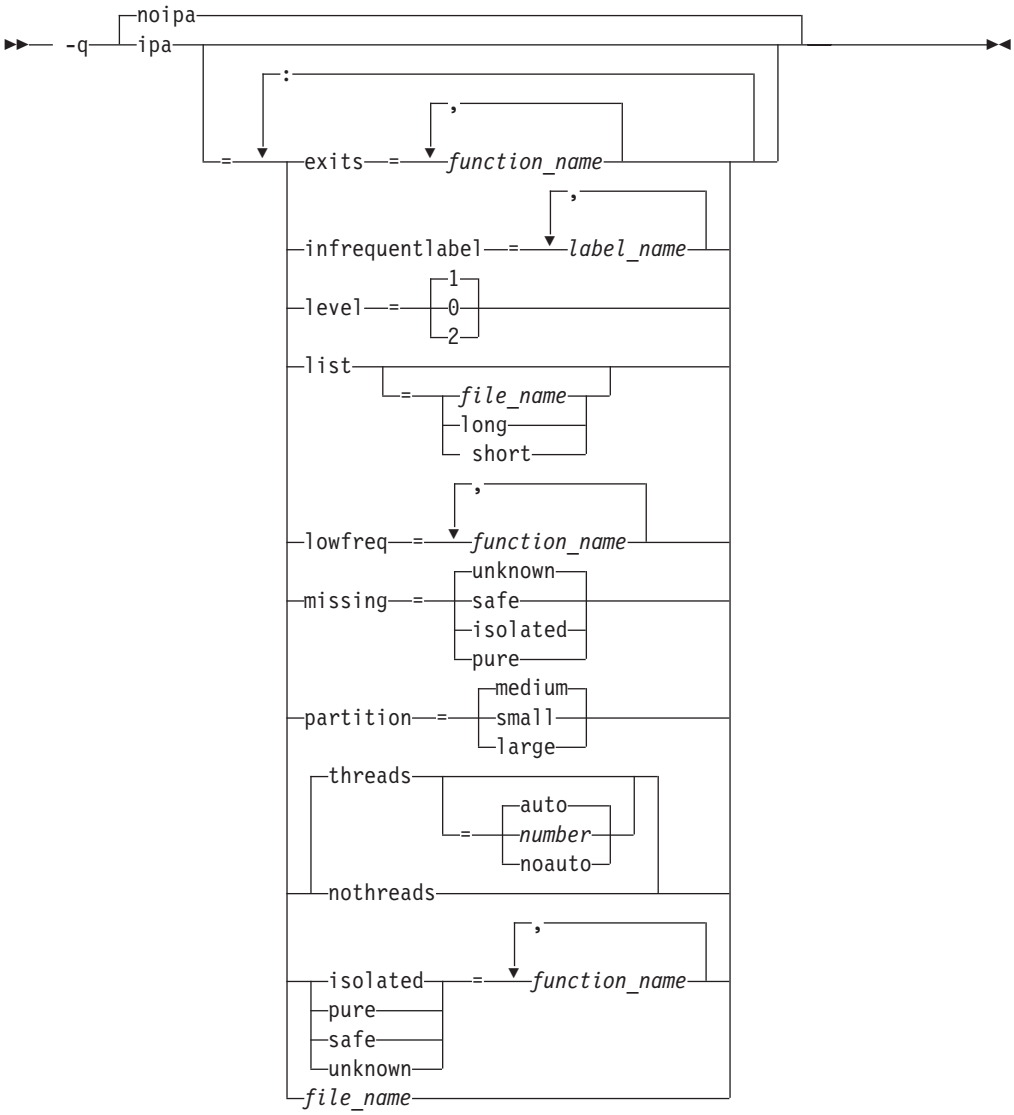
IPA is a two-step process: the first step, which takes place during compilation, consists of performing an initial analysis and storing interprocedural analysis information in the object file. The second step, which takes place during linking, and causes a complete recompilation of the entire application, 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**

Parameters

You can specify the following parameters 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**.

You can specify the following parameters during a combined compilation and link stepin the same compiler invocation, or during a separate link step only:

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.

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 as follows:

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.

To generate data reorganization information, specify the optimization level **-qipa=level=2** or **-O5** together with **-qreport**. During the IPA link phase, the data reorganization messages for program variable data are produced in the data reorganization section of the listing file. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

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 suboptions:

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 suboptions:

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**.

partition
Specifies the size of each program partition created by IPA during pass 2. Valid suboptions are one of the following suboptions:

- **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.

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 one of the following suboptions:

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 shown as follows:

```
# ... comment
attribute{, attribute} = name{, name}
missing = attribute{, attribute}
exits = name{, name}
lowfreq = name{, name}
list [ = file-name | short | long ]
level = 0 | 1 | 2
partition = small | medium | large
```

where *attribute* is one of:
- exits
- lowfreq
- unknown
- safe
- isolated
- pure

Usage

Specifying **-qipa** automatically sets the optimization level to **-O2**. For additional performance benefits, you can also specify the **-qinline** 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**.

You can link objects created with different releases of the compiler, but you must ensure that you use a linker that is at least at the same release level as the newer of the compilers used to create the objects being linked.

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:

```
bgxlc -c *.c -qipabgxlc
 -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:

```
bgxlc -c *.c -qipa=noobject
bgxlc -c *.o -qipa=lowfreq=user_trace[123]:exit=user_abort
```

Related information

- “-qinline” on page 149
- “-qisolated_call”
- “-qlibmpi” on page 189
- “#pragma execution_frequency” on page 318
- “-S” on page 238
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*
- Runtime environment variables

-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:

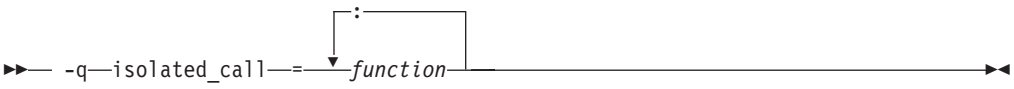
- Accessing a volatile object

- 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

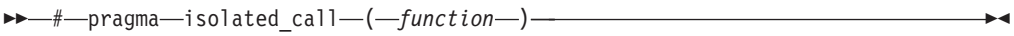
Marking a function as isolated indicates to the optimizer that external and static variables cannot be changed by the called function and that pessimistic references to storage can be deleted from the calling function where appropriate. Instructions can be reordered with more freedom, resulting in fewer 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



Pragma syntax



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. C++ 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:

```
bgxlc 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 136
- "The const function attribute" and "The pure function attribute" in the *XL C/C++ Language Reference*

-qkeepparm

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

When used with **-O2** or higher optimization, specifies whether procedure 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 **-qkeepparm** is in effect, parameters are stored on the stack even when optimization is enabled. When **-qnokeepparm** is in effect, parameters are removed from the stack if this provides an optimization advantage.

Syntax

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 207

-qkeyword

Category

Language element control

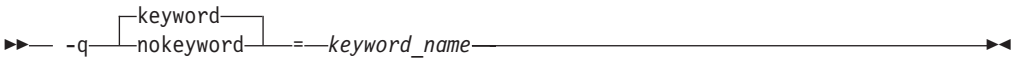
Pragma equivalent

None

Purpose

Controls whether the specified name is treated as a keyword or as an identifier whenever it appears in your program source.

Syntax



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.

On Blue Gene/Q, **-qnokeyword=vector4double** disables the `vector4double` keyword.

C++

This option can be used with all C++ built-in keywords.

C++0x This option can be used with the following new keywords introduced by the C++0x standard:

Table 23. New keywords introduced by the C++0x standard

Keyword	Feature	-qlanglvl suboption
`constexpr`	Generalized constant expressions[1]	**-qlanglvl=[no]constexpr**
`decltype`	"The decltype(expression) type specifier (C++0x)"	**-qlanglvl=[no]decltype**
`static_assert`	"static_assert declaration (C++0x)"	**-qlanglvl=[no]static_assert**

Note:

1. In XL C/C++ V12.1, this feature is a partial implementation of what is defined in the C++0x standard.

These features introduce new keywords to the C++0x standard in addition to new semantics. To enable each feature semantically, the associated *keyword_name* (`constexpr`, `decltype`, or `static_assert`) must be recognized as a keyword by the compiler.

You can use the **-qlanglvl** suboption of each feature or the **-qlanglvl=extended0x** group option to enable both the feature and the associated keyword. If you enable the **-qkeyword=***keyword_name* option, the compiler reserves the *keyword_name* as a keyword, but the associated feature is not enabled automatically.

C++0x

C++

C This option can also be used with the following C keywords:

- asm
- inline
- restrict
- typeof

Note: `asm` is not a keyword when the **-qlanglvl** option is set to **stdc89** or **stdc99**.

C

Predefined macros

- __VECTOR4DOUBLE__ is defined to 1 by default.
- 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

You can reinstate `vector4double` with the following invocation:

```
bgxlc++ -qkeyword=vector4double
```

C++ You can reinstate `bool` with the following invocation:

```
bgxlc++ -qkeyword=bool
```

C++

C You can reinstate `typeof` with the following invocation:

```
bgxlc -qkeyword=typeof
```

C

Related information

- Vector types (IBM extension)
- "-qasm" on page 71
- "-qrestrict (C only)" on page 233

-l

Category

Linking

Pragma equivalent

None.

Purpose

Searches for the specified library file, lib*key*.so, and then lib*key*.a for dynamic linking, or just for lib*key*.a for static linking.

Syntax

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:

```
bgxlc myprogram.c -lmylibrary -L/usr/mylibdir
```

Related information

- "-L"
- "Specifying compiler options in a configuration file" on page 7

-L

Category

Linking

Pragma equivalent

None.

Purpose

At link time, searches the directory path for library files specified by the **-l** option.

Syntax

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

Paths specified with the **-L** compiler option are only searched at link time. To specify paths that should be searched at run time, use the **-R** option.

If the **-L***directory* 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:

```
bgxlc myprogram.c -lspfiles -L/usr/tmp/old
```

Related information

- "-l" on page 163
- "-R" on page 229

-qlanglvl

This topic includes the following information:

Category

Language element control

Pragma equivalent

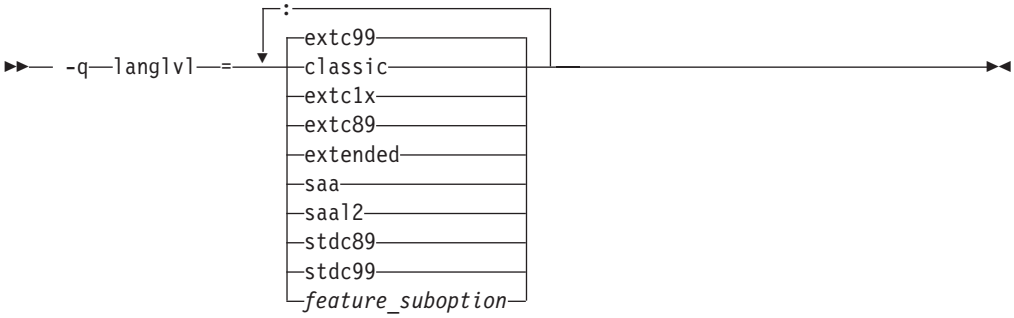
C #pragma options langlvl, #pragma langlvl

Purpose

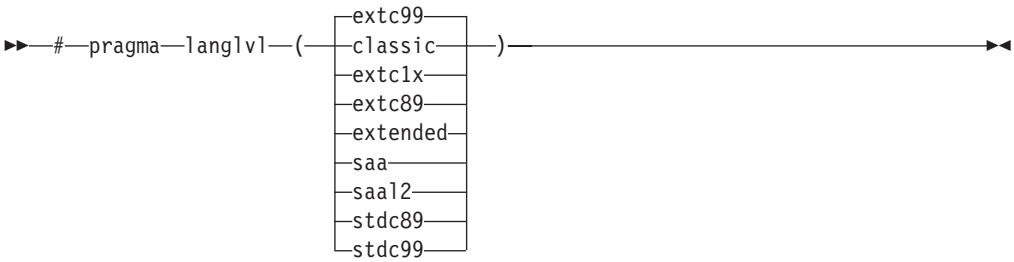
Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.

Syntax

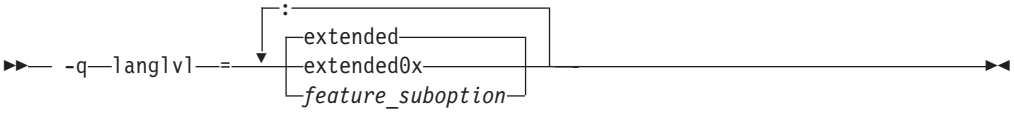
-qlanglvl syntax — C



#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 **bgxlc** and related invocation commands
  - **-qlanglvl=extended:noucs** for the **bgcc** and related invocation commands
  - **-qlanglvl=stdc89:noucs** for the **bgc89** and related invocation commands
  - **-qlanglvl=stdc99:ucs** for the **bgc99** and related invocation commands
- C++ The suboptions and their default settings for different language levels (**compat366**, **extended** (C++), and **extended0x**) are listed in Table 24. The default setting On means that the suboption is enabled; otherwise, the default setting Off means that the suboption is disabled.

Table 24. Default Settings of suboptions for different language levels

Options	Language levels		
	compat366	**extended (C++)**	**extended0x**
-qlanglvl=anonstruct | noanonstruct	Off	On	On
-qlanglvl=anonunion | noanonunion	On	On	On
-qlanglvl=ansifor | noansifor	Off	On	On
-qlanglvl=ansisinit | noansisinit	On	On	On
C++0x -qlanglvl=autotypededuction | noautotypededuction	Off	Off	On
-qlanglvl=c1xnoreturn | noc1xnoreturn	Off	On	On
-qlanglvl=c99__func__ | noc99__func__	Off	On	On
-qlanglvl=c99complex | noc99complex	Off	Off	Off
-qlanglvl=c99complexheader | noc99complexheader	Off	Off	Off
-qlanglvl=c99compoundliteral | noc99compoundliteral	Off	On	On
-qlanglvl=c99hexfloat | noc99hexfloat	Off	On	On
C++0x -qlanglvl=c99longlong | noc99longlong	Off	Off	On
C++0x -qlanglvl=c99preprocessor | noc99preprocessor	Off	Off	On
-qlanglvl=c99vla | noc99vla	Off	On	On
IBM -qlanglvl=compatrvaluebinding | nocompatrvaluebinding	Off	Off	Off
-qlanglvl=complexinit | nocomplexinit	Off	On	On
C++0x -qlanglvl=constexpr | noconstexpr	Off	Off	On

Table 24. Default Settings of suboptions for different language levels (continued)

Options	Language levels		
	compat366	extended (C++)	extended0x
C++0x -qlanglvl=decltype | nodecltype	Off	Off	On
C++0x -qlanglvl=delegatingctors | nodelegatingctors	Off	Off	On
-qlanglvl=dependentbaselookup | nodependentbaselookup	On	On	Off
-qlanglvl=emptystruct | noemptystruct	On	On	On
C++0x -qlanglvl=explicitconversionoperators | noexplicitconversionoperators	Off	Off	On
C++0x -qlanglvl=extendedfriend | noextendedfriend	Off	Off	On
C++0x IBM -qlanglvl=extendedintegersafe | noextendedintegersafe	Off	Off	Off
C++0x -qlanglvl=externtemplate | noexterntemplate	Off	On	On
-qlanglvl=FileScopeConstExternLinkage | noFileScopeConstExternLinkage	Off	Off	Off
C++0x -qlanglvl=inlinenamespace | noinlinenamespace	Off	Off	On
-qlanglvl=gnu_assert | nognu_assert	Off	On	On
-qlanglvl=gnu_complex | nognu_complex	Off	Off	Off
-qlanglvl=gnu_computedgoto | nognu_computedgoto	Off	On	On
-qlanglvl=gnu_explicitregvar | nognu_explicitregvar	Off	On	On
-qlanglvl=gnu_externtemplate | nognu_externtemplate	Off	On	On
-qlanglvl=gnu_labelvalue | nognu_labelvalue	Off	On	On
-qlanglvl=gnu_locallabel | nognu_locallabel	Off	On	On
-qlanglvl=gnu_include_next | nognu_include_next	On	On	On
-qlanglvl=gnu_membernamereuse | nognu_membernamereuse	Off	On	On
-qlanglvl=gnu_suffixij | nognu_suffixij	Off	On	On
-qlanglvl=gnu_varargmacros | nognu_varargmacros	Off	On	On
-qlanglvl=gnu_warning | nognu_warning	Off	On	On
-qlanglvl=illptom | noillptom	On	On	On
-qlanglvl=implicitint | noimplicitint	On	On	On
-qlanglvl=newexcp | nonewexcp	Off	Off	Off
-qlanglvl=offsetnonpod | nooffsetnonpod	On	On	On

Table 24. Default Settings of suboptions for different language levels (continued)

Options	Language levels		
	compat366	**extended (C++)**	**extended0x**
-qlanglvl=olddigraph | noolddigraph	Off	Off	Off
-qlanglvl=oldfriend | nooldfriend	On	On	Off
-qlanglvl=oldmath | nooldmath	On	Off	Off
-qlanglvl=oldtempacc | nooldtempacc	On	On	On
-qlanglvl=oldtmplalign | nooldtmplalign	On	Off	Off
-qlanglvl=oldtmplspec | nooldtmplspec	On	On	On
-qlanglvl=redefmac | noredefmac	Off	Off	Off
C++0x -qlanglvl=referencecollapsing | noreferencecollapsing	Off	Off	On
C++0x -qlanglvl=rightanglebracket | norightanglebracket	Off	Off	On
C++0x -qlanglvl=rvaluereferences | norvaluereferences	Off	Off	On
C++0x -qlanglvl=scopedenum | noscopedenum	Off	Off	On
C++0x -qlanglvl=static_assert | nostatic_assert	Off	Off	On
IBM -qlanglvl=tempsaslocals | notempsaslocals	Off	Off	Off
IBM -qlanglvl=textafterendif | notextafterendif	Off	Off	Off
-qlanglvl=trailenum | notrailenum	On	On	On
-qlanglvl=typedefclass | notypedefclass	On	On	On
-qlanglvl=noucs | nonoucs	Off	Off	Off
-qlanglvl=varargmacros | novarargmacros	Off	On	On
C++0x -qlanglvl=variadic[templates] | novariadic[templates]	Off	Off	On
-qlanglvl=zeroextarray | nozeroextarray	Off	On	On

Parameters for C language programs

C 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 */

#dfine\
M 1          /* equivalent to #dfine M 1, even
                though #dfine 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 **-qcpluscmt** 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.

C1X

extc1x

Compilation is based on the C1X standard, invoking all the currently supported C1X features and other implementation-specific language extensions. For more information about these C1X features, see Extensions for C1X compatibility in the *XL C/C++ Language Reference*.

Note: C1X is a new version of the C programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C1X standard is complete, including the support of a new C standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C1X standard and therefore they should not be relied on as a stable programming interface.

Note: C1X has been ratified and published as ISO/IEC 9899:2011. All references to C1X in this document are equivalent to the ISO/IEC 9899:2011 standard. Corresponding information, including programming interfaces, will be updated in a future release.

C1X

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.

The **-qlanglvl** suboption parameters for individual C features are listed as follows:

feature_suboption
feature_suboption in the syntax diagram represents a colon-separated list of the C options. They can be any of the following options:

Note: When multiple **-qlanglvl** group options and suboptions are specified for one individual C feature, the last one takes effect.

IBM **textafterendif | notextafterendif**
Specifies whether to suppress the warning message that is emitted when you are porting code from a compiler that allows extra text after `#endif` or `#else` to the IBM XL C/C++ compiler. The default option is **-qlanglvl=notextafterendif**, indicating that a message is emitted if `#else` or `#endif` is followed by any extraneous text. However, when the language level is **classic**, the default option is **-qlanglvl=textafterendif**, because this language level already allows extra text after `#else` or `#endif` without generating a message. IBM

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.

Parameters for C++ language programs

C++ The following are the **-qlanglvl** group option parameters for corresponding C++ language levels:

extended
Compilation is based on the ISO C++ standard, with some differences to accommodate extended language features.

C++0x **extended0x**
Compilation is based on the C++0x standard, invoking most of the C++ features and all the currently-supported C++0x features. Table 24 on page 166

provides details about the supported features. For more information about C++0x features, see "Extensions for C++0x compatibility" in the *XL C/C++ Language Reference*.

Note: C++0x is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++0x standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++0x standard and therefore they should not be relied on as a stable programming interface.

Note: C++0x has been ratified and published as ISO/IEC 14882:2011. All references to C++0x in this document are equivalent to the ISO/IEC 14882:2011 standard. Corresponding information, including programming interfaces, will be updated in a future release.

The following are the **-qlanglvl** suboption parameters for individual C++ features.

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:

Note: When multiple **-qlanglvl** group options and suboptions are specified for one individual C++ feature, the last one takes effect.

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 previous versions of VisualAge® C++ and predecessor products, and 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 earlier versions of VisualAge C++ and predecessor products, and 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; specify **-qlanglvl=noansisinit** for compatibility with earlier versions of VisualAge C++ and predecessor products.

C++0x **autotypededuction| noautotypededuction**

Controls whether the auto type deduction feature is enabled. When you specify the **-qlanglvl=autotypededuction** option, the auto type deduction feature is enabled, with which you no longer need to specify a type while declaring a variable. Instead, the compiler deduces the type of an auto variable from the type of its initializer expression.

You can also use the **-qlanglvl=autotypededuction** option to control the trailing return type feature. This feature is useful when declaring the following types of templates and functions:

- Function templates or member functions of class templates with return types that depend on the types of the function arguments
- Functions or member functions of classes with complicated return types
- Perfect forwarding functions

The **-qlanglvl=autotypededuction** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the auto type deduction feature.

The default option is **-qlanglvl=noautotypededuction**.

c1xnoreturn | noc1xnoreturn

Enables or disables support of the `_Noreturn` function specifier.

The **-qlanglvl=c1xnoreturn** option is included in group options **-qlanglvl=extended** and **-qlanglvl=extended0x**, so you can also use these group options to enable the `_Noreturn` function specifier.

The default option is **-qlanglvl=noc1xnoreturn**.

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.

C++0x **c99longlong** | **noc99longlong**
Controls whether the C99 `long long` feature is enabled. When you specify the **-qlanglvl=c99longlong** option, the C++ compiler provides the C99 `long long` feature, which improves source compatibility between the C and C++ languages.

The **-qlanglvl=c99longlong** option conflicts with the **-qlonglong** option. If you specify both these two options, the **-qlonglong** option is ignored. For more information about the **-qlonglong** option, see "-qlonglong" on page 197.

The **-qlanglvl=c99longlong** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the C99 `long long` feature.

The default option is **-qlanglvl=noc99longlong**.

C++0x **c99preprocessor** | **noc99preprocessor**
Controls whether the C99 preprocessor features adopted in C++0x are enabled. When **-qlanglvl=c99preprocessor** is in effect, the C99 and C++0x compilers provide a more common preprocessor interface, which can ease porting C source files to the C++ compiler and avoid preprocessor compatibility issues.

The default option is **-qlanglvl=noc99preprocessor**.

Note: Specifying **-qlanglvl=c99preprocessor** implicitly sets **-qlanglvl=varargmacros**. Also, specifying **-qlanglvl=noc99preprocessor** implicitly sets **-qlanglvl=novarargmacros**.

The **-qlanglvl=c99preprocessor** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the C99 preprocessor feature.

c99vla | **noc99vla**
Enables or disables support for C99-type variable length arrays.

IBM **compatrvaluebinding** | **nocompatrvaluebinding**
The C++ Standard (2003) indicates that an rvalue can only be bound to a const non-volatile lvalue reference. Non-compliant compilers might allow a non-const or volatile lvalue reference to be bound to an rvalue. When you are porting code to IBM XL C/C++ compiler, you can specify this option to instruct the compiler to allow a non-const or volatile lvalue reference to bind to an rvalue of a user-defined type where an initializer is not required. IBM

C++0x If both the **-qlanglvl=compatrvaluebinding** and **-qlanglvl=rvaluereferences** options are in effect, the compiler issues an error message. C++0x

C++ **complexinit | nocomplexinit**
Controls whether the C++ compiler uses the C1X style initialization of the C99 complex types.

The **-qlanglvl=complexinit** option is included in the group options **-qlanglvl=extended** and **-qlanglvl=extended0x**, so you can also use these group options to enable the initialization of complex types. In this case, specify **-qlanglvl=c99complexheader** so that correct header files can be used by the compiler.

The default option is **-qlanglvl=complexinit**. C++

C++0x **constexpr | noconstexpr**
Controls whether the generalized constant expressions feature is enabled. When you specify the **-qlanglvl=constexpr** option, the compiler extends the expressions permitted within constant expressions. A constant expression is one that can be evaluated at compile time.

The **-qlanglvl=constexpr** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the generalized constant expressions feature.

The default option is **-qlanglvl=noconstexpr**.

Note: In XL C/C++ V12.1, this feature is a partial implementation of what is defined in the C++0x standard. Here are the implementation details:

- Support valid syntax using the `constexpr` keyword.
- Support valid definitions of `constexpr` functions whose arguments and return type are of built-in types.
- Diagnostic messages might not be issued for invalid `constexpr` function definitions. A function call to an invalid `constexpr` function definition might prevent the compiler from evaluating the result, and an error is issued depending on the context where a constant expression is required.

C++0x **decltype | nodecltype**
Controls whether the `decltype` feature is enabled. With this feature, you can get a type that is based on the resultant type of a possibly type-dependent expression. To enable this feature, you can specify the **-qlanglvl=decltype** option.

The **-qlanglvl=decltype** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the `decltype` feature.

The default option is **-qlanglvl=nodecltype**.

C++0x **delegatingctors | nodelegatingctors**
Controls whether the delegating constructors feature is enabled. With this feature, you can concentrate on common initializations and post initializations in one constructor, which can make programs more readable and maintainable. To enable this feature, you can specify the **-qlanglvl=delegatingctors** option.

The **-qlanglvl=delegatingctors** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the delegating constructors feature.

The default option is **-qlanglvl=nodelegatingctors**.

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.

Note: The default option is **-qlanglvl=noDependentBaseLookup** at the C++0x language level.

The following example shows code that does not compile with **-qlanglvl=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 **-qlanglvl=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;
}
```

empty_struct | noempty_struct

This option instructs the compiler to tolerate empty member declarations in structs. Empty member declaration in structs is not allowed. For example, when **-qlanglvl=noemptystruct** is in effect, the following example will be rejected by the compiler:

```
struct S {
      ;  // this line is ill-formed
};
```

The default is **-qlanglvl=noemptystruct**.

C++0x **explicitconversionoperators | noexplicitconversionoperators**
Controls whether the explicit conversion operators feature is enabled. When you specify the **-qlanglvl=explicitconversionoperators** option, you can apply the `explicit` function specifier to the definition of a user-defined conversion function, and thus inhibit unintended implicit conversions through the user-defined conversion function.

The **-qlanglvl=explicitconversionoperators** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the explicit conversion operators feature.

The default option is **-qlanglvl=noexplicitconversionoperators**.

C++0x **extendedfriend | noextendedfriend**
Controls whether the extended friend declarations feature is enabled. When you specify the **-qlanglvl=extendedfriend** option, rules governing friend declarations are relaxed as follows:

- Template parameters, `typedef` names, and basic types can be declared as friends.
- The class-key in the context for friend declarations is no longer necessary in C++0x.

The **-qlanglvl=extendedfriend** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the extended friend declarations feature.

The default option is **-qlanglvl=noextendedfriend**.

Note: **-qlanglvl=extendedfriend** is incompatible with the **-qlanglvl=oldfriend** option. When **-qlanglvl=extendedfriend** is in effect, the **-qlanglvl=oldfriend** option is ignored and the setting of **-qlanglvl=[no]oldfriend** is **-qlanglvl=nooldfriend**.

C++0x IBM **extendedintegersafe | noextendedintegersafe**
With this option, if a decimal integer literal that does not have a suffix containing `u` or `U` cannot be represented by the `long long int` type, you can decide whether to use the `unsigned long long int` type to represent the literal or not.

This option takes effect only when the **-qlanglvl=c99longlong** option is specified, otherwise, the compiler issues a warning message to indicate that the option is ignored. When you specify both the **-qlanglvl=c99longlong** and **-qlanglvl=extendedintegersafe** options, if a decimal integer literal that does not have a suffix containing `u` or `U` cannot be represented by the `long long int` type, the compiler issues an error message stating that the value of the literal is out of range.

The default option is **-qlanglvl=noextendedintegersafe** in all the language levels.

C++0x **externtemplate | noexterntemplate**
Controls whether the explicit instantiation declarations feature is enabled. With this feature, you can suppress the implicit instantiations of a template specialization or its members. To enable this feature, you can specify the **-qlanglvl=externtemplate** option, which is the default option.

The **-qlanglvl=externtemplate** option is included in the group options of **-qlanglvl=extended** and **-qlanglvl=extended0x**, so you can use these two group options to enable this feature.

The following table lists options that interact with the **-qlanglvl=externtemplate** option:

Table 25. Options that interact with **-qlanglvl=externtemplate**

Option	Description
-qtemplateregistry, -qtempinc	Explicit instantiation declarations remain effective. Referenced specializations that are the subjects of explicit instantiation declarations, but not the subjects of explicit instantiation definitions in a translation unit are not instantiated from or because of that translation unit.

The following table lists IBM language extensions that interact with the **-qlanglvl=externtemplate** option:

Table 26. IBM language extensions that interact with **-qlanglvl=externtemplate**

IBM language extension	Description
#pragma instantiate	This pragma is semantically the same as an explicit instantiation definition.
#pragma do_not_instantiate	This pragma provides a subset of the functionality of the explicit instantiation declarations which is introduced in the C++0x standard. It is provided for backwards compatibility purposes only. New applications can use explicit instantiation declarations.
#pragma hashome, #pragma ishome	This pragma causes the generation of the virtual function table (VFT) for a class template specialization irrespective of explicit instantiation declarations of the specialization.

The **-qlanglvl=[no]externtemplate** option replaces the deprecated **-qlanglvl=[no]gnu_externtemplate** option. Use the **-qlanglvl=[no]externtemplate** option in your applications.

FileScopeConstExternLinkage | noFileScopeConstExternLinkage
Controls whether the file scope of constant variables has internal or external linkage when the static or extern keyword is not specified.

When **-qlanglvl=FileScopeConstExternLinkage** is in effect, all file scope constant variables are marked as externally visible. Otherwise, all file scope constant variables are marked as static.

The default is **-qlanglvl=noFileScopeConstExternLinkage**.

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*.

Note: The option **-qlanglvl=[no]gnu_externtemplate** is deprecated in XL C/C++ V12.1; you can use the option **-qlanglvl=[no]externtemplate** instead.

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::func`, in the old style:

```
struct C {
void func(int);
};

void (C::*p) (int) = C::func;
```

This is an extension to standard C++ and gives behavior that is designed to be compatible with earlier versions of VisualAge C++ and its predecessor products, and 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 func(int);
};

void (C::*p) (int) = &C::func;
```

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

C++0x C++0x has removed the use of auto as a storage class specifier. In C++0x, the keyword auto is used as a type specifier. The compiler deduces the type of an auto variable from the type of its initializer expression. For more information, see "The auto type specifier (C++0x)" in the *XL C/C++ Language Reference*.

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 earlier versions of VisualAge C++ and predecessor products, and 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;
}
```

C++0x **inlinenamespace | noinlinenamespace**

Controls whether inline namespace definitions are enabled, which are namespace definitions preceded by an initial inline keyword. A namespace so defined is an inline namespace. When you specify the **-qlanglvl=inlinenamespace** option, members of the inline namespace can be defined and specialized as if they were also members of the enclosing namespace.

The **-qlanglvl=inlinenamespace** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the inline namespace definitions feature.

The default option is **-qlanglvl=noinlinenamespace**.

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 VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, V3.5, and 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++ and the extended C++ language level supported by previous versions of VisualAge C++ and predecessor products.

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 earlier versions of VisualAge C++ and predecessor products, and 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;
```

Note: **-qlanglvl=oldfriend** is incompatible with the **-qlanglvl=extendedfriend** option. When **-qlanglvl=extendedfriend** is in effect, the **-qlanglvl=oldfriend** option is ignored and the setting of **-qlanglvl=[no]oldfriend** is **-qlanglvl=nooldfriend**.

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 VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, V3.5, and 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 func() {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 = func(); // use the copy of a C object
//                              created by func()
}
```

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
};
```

Specify **-qlanglvl=oldtmplalign** for compatibility with VisualAge for C++ V4.0 and predecessor products.

oldtmplspec | nooldtmplspec

Controls whether template specializations that do not conform to the C++ standard are allowed. When the default, **-qlanglvl=oldtmplspec**, 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 VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, V3.5, and Microsoft Visual C++.

Specify **-qlanglvl=nooldtmplspec** 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.

C++0x **referencecollapsing** | **noreferencecollapsing**
Controls whether the reference collapsing feature is enabled. To enable this feature, specify the **-qlanglvl=referencecollapsing** option.

The **-qlanglvl=referencecollapsing** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the reference collapsing feature.

When the **-qlanglvl=rvaluereferences** option is in effect, but the **-qlanglvl=referencecollapsing** option is not in effect, the compiler behaves as if the **-qlanglvl=referencecollapsing** option were specified.

The default option is **-qlanglvl=noreferencecollapsing**.

C++0x **rightanglebracket** | **norightanglebracket**
Controls whether the right angle bracket feature is enabled. To enable this feature, you can specify the **-qlanglvl=rightanglebracket** option.

The **-qlanglvl=rightanglebracket** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the right angle bracket feature.

The default option is **-qlanglvl=norightanglebracket**.

C++0x **rvaluereferences** | **norvaluereferences**
Controls whether the rvalue references feature is enabled. To enable this feature, specify the **-qlanglvl=rvaluereferences** option.

The **-qlanglvl=rvaluereferences** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the rvalue references feature.

If both the **-qlanglvl=compatrvaluebinding** and **-qlanglvl=rvaluereferences** options are in effect, the compiler issues an error message.

The default option is **-qlanglvl=norvaluereferences**.

C++0x **scopedenum** | **noscopedenum**
Controls whether the scoped enumeration feature is enabled. To enable this feature, you can specify the **-qlanglvl=scopedenum** option.

The **-qlanglvl=scopedenum** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the scoped enumeration feature.

The default option is **-qlanglvl=noscopedenum**.

C++0x **static_assert** | **nostatic_assert**
Controls whether the static assertions feature is enabled. When **-qlanglvl=static_assert** is in effect, this feature can be used to produce compile-time assertions for which a severe error message is issued on failure.

-qlanglvl=static_assert is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the static assertions feature.

The default is **-qlanglvl=nostatic_assert**.

IBM **tempsaslocals** | **notempsaslocals**
The C++ Language Standard describes the lifetime of temporaries in section Temporary Object [class.temporary]. When you are porting an application from a compiler that implements late temporary destruction, you might need to extend the lifetime of C++ temporaries beyond which is specified in the C++ Language Standard. This option extends the lifetime of temporaries to reduce migration difficulty.

textafterendif | **notextafterendif**
Specifies whether to suppress the warning message that is emitted when you are porting code from a compiler that allows extra text after `#endif` or `#else` to IBM XL C/C++ compiler. The default option is **-qlanglvl=notextafterendif**, indicating that a message is emitted if `#else` or `#endif` is followed by any extraneous text. IBM

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, for compatibility with earlier versions of VisualAge for C++ and predecessor products.

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 | **novarargmacros**
Enables or disables support for C99-style variable argument lists in function-like macros.

Note: Specifying **-qlanglvl=c99preprocessor** implicitly set **-qlanglvl=varargmacros**. Vice versa, specifying **-qlanglvl=noc99preprocessor** implicitly set **-qlanglvl=novarargmacros**.
For details of this feature, see "Function-like macros" in the *XL C/C++ Language Reference*.

C++0x **variadic[templates]** | **novariadic[templates]**
Controls whether the variadic templates feature is enabled. With this feature, you can define class and function templates that have any number (including zero) of parameters. To enable this feature, you can specify the **-qlanglvl=variadic[templates]** option. The word *templates* included in the brackets is optional. If you specify only the **-qlanglvl=variadic** option, the compiler assumes that the **-qlanglvl=variadictemplates** option is specified.

The **-qlanglvl=variadic[templates]** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the variadic templates feature.

The default option is **-qlanglvl=novariadic[templates]**.

zeroextarray | nozeroextarray

Controls whether you can use zero-extent arrays as the last nonstatic data member in a structure definition. When the default, **-qlanglvl=zeroextarray**, is in effect, you can use arrays with zero elements. The following statement declares a zero-extent array `a`.

```
struct S1 { char a[0]; };
```

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++ or with the ANSI language level supported by previous versions of VisualAge C++ and predecessor products.

Usage

C++ 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

Note: In the C++0x language level, if you use the **no** form of a suboption to disable the C++0x meaning of `decltype` or `static_assert`, the compiler emits syntax errors but no diagnostic message if the user happens to use the C++0x syntax of `decltype` or `static_assert`.

C 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 387 for a list of macros that are predefined by **-qlanglvl** suboptions.

Related information

- “-qsuppress” on page 266
- "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*

-qldbl128

Category

Floating-point and integer control

Pragma equivalent

#pragma options [no]ldbl128

Purpose

Increases the size of `long double` types from 64 bits to 128 bits.

Syntax

Defaults

-qldbl128

Usage

The **#pragma options** directive must appear before the first C or C++ statement in the source file, and the option applies to the entire file.

Predefined macros

- __LONGDOUBLE128 and __LONG_DOUBLE_128__ are defined to 1 when **-qldbl128** is in effect; otherwise, they are undefined.
- __LONGDOUBLE64 is defined to 1 when **-qnoldbl128** is in effect; it is undefined when **-qldbl128** is in effect.

Examples

To compile `myprogram.c` so that long double types are 128 bits, enter:

```
bgxlc myprogram.c -qldbl128
```

-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

Defaults

-qlib

Usage

Using **-qnolib** 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 **-qnolib**, 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:

```
bgxlc myprogram.c -qnolib -lxlopt
```

Related information

- "-qcrt" on page 90

-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

Defaults

-qnolibansi

Predefined macros

C++ __LIBANSI__ is defined to 1 when **libansi** is in effect; otherwise, it is not defined.

-qlibmpi

Category

"Optimization and tuning" on page 54

Pragma equivalent

None

Purpose

Asserts that all functions with Message Passing Interface (MPI) names are in fact MPI functions and not a user function with different semantics.

Syntax

Defaults

-qnolibmpi

Usage

MPI is a library interface specification for message passing. It addresses the message-passing parallel programming model in which data is moved from the address space of one process to another through cooperative operations. For details about MPI, see the Message Passing Interface Forum.

-qlibmpi allows the compiler to generate better code because it knows about the behavior of a given function, such as whether or not it has any side effects.

When you use **-qlibmpi**, the compiler assumes that all functions with the name of an MPI library function are in fact MPI functions. **-qnolibmpi** makes no such assumptions.

Note: You cannot use this option if your application contains your own version of the library function that is incompatible with the standard one.

Predefined macros

None.

Examples

To compile `myprogram.c`, enter the following command:

```
bgxlc -O5 myprogram.c -qlibmpi
```

Related information

- Message Passing Interface Forum
- "-qipa" on page 151

-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

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:

```
bgxlc myprogram.c -o testing -qlinedebug
```

Related information

- "-g" on page 121
- "-O, -qoptimize" on page 207

-qlist

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]list

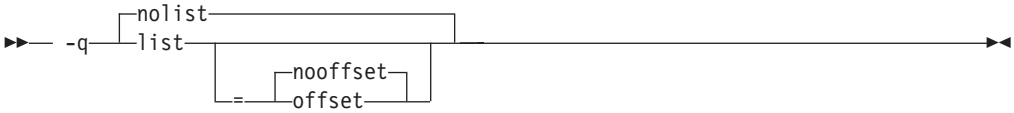
Purpose

Produces a compiler listing file that includes an object 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 17.

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

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:

```
bgxlc myprogram.c -qlist
```

Related information

- "-qlistopt" on page 195
- "-qprint" on page 223
- "-qsource" on page 251

-qlistfmt

Category

Listings, messages, and compiler information

@PROCESS

None.

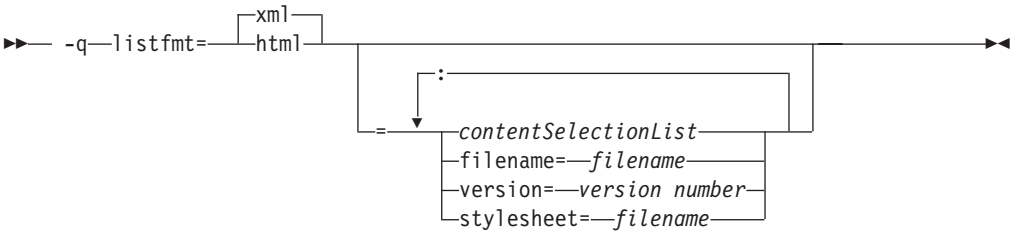
Pragma equivalent

None.

Purpose

Creates an XML or HTML report to assist with finding optimization opportunities.

Syntax



Defaults

This option is off by default. If no *contentSelectionList* options are selected in their positive form, all available report information is produced. For example, specifying **-qlistfmt=xml** is equivalent to **-qlistfmt=xml=all**.

Parameters

The following list describes **-qlistfmt** parameters:

xml | **html**
Indicates that the report should be generated in XML or HTML format. If an XML report has been generated before, you can convert the report to the HTML format using the **genhtml** command. For more information about this command, see "genhtml command" on page 195.

contentSelectionList
The following suboptions provide a filter to limit the type and quantity of information in the report:

data | **nodata**
Produces data reorganization information.

inlines | **noinlines**
Produces inlining information.

transforms | **notransforms**
Produces loop transformation information.

all
Produces all available report information.

none
Does not produce a report.

filename
Specifies the name of the report file. One file is produced during the compile phase, and one file is produced during the IPA link phase. If no filename is specified, a file with the suffix `.xml` or `.html` is generated in a way that is consistent with the rules of name generation for the given platform. For example, if compiling `foo.c` the generated XML files are `foo.xml` from the compile step and `a.xml` from the link step.

Note: If you compile and link in one step and use this suboption to specify a file name for the report, the information from the IPA link step will overwrite the information generated during the compile step.

The same will be true if you compile multiple files using the `filename` suboption. The compiler creates an report for each file so the report of the last file compiled will overwrite the previous reports. For example,

```
bgxlc -qlistfmt=xml=all:filename=abc.xml -O3 myfile1.c myfile2.c myfile3.c
```

will result in only one report, `abc.xml` based on the compilation of the last file `myfile3.c`

stylesheet
Specifies the name of an existing XML stylesheet for which an `xml-stylesheet` directive is embedded in the resulting report. The default behavior is to not include a stylesheet. The stylesheet shipped with XL C/C++ is `xlstyle.xsl`. This stylesheet renders the XML to an easily read format when viewed using a browser that supports XSLT.

To view the XML report created with the **stylesheet** suboption, you must place the actual stylesheet (`xlstyle.xsl`) and the XML message catalogue (`XMLMessages-`*locale*`.xml` where *locale* refers to the locale set on the compilation machine) in the path specified by the **stylesheet** suboption. For example, if `a.xml` is generated with **stylesheet=xlstyle.xsl**, `xlstyle.xsl` and

XMLMessages-*locale*.xml must be in the same directory as a.xml, before you can properly view a.xml with a browser. The message catalogs and stylesheet are installed in the /opt/ibmcmp/vacpp/bg/12.1/listings/ directory.

version
Specifies the major version of the content that is emitted. If you have written a tool that requires a certain version of this report, you should specify the version. IBM XL C/C++ for Blue Gene/Q, V12.1 creates reports at XML v1.1, so if you have written a tool to consume these reports, specify version=v1.

Usage

The information produced in the report by the **-qlistfmt** option depends on which optimization options are used to compiler the program.

- When used with an option that enables inlining such as **-qinline**, the report shows which functions were inlined and why others were not inlined.
- When used with an option that enables loop unrolling, the report contains a summary of how program loops are optimized. The report also includes diagnostic information to show why specific loops cannot be vectorized. For **-qlistfmt** to generate information about loop transformations, you must also specify at least one of the following options:
  - **-qsimd=auto**
  - **-qsmp**
  - **-O5**
  - **-qipa=level=2**
- When used with an option that enables parallel transformations, the report contains information about parallel transformations. For **-qlistfmt** to generate information about parallel transformations or parallel performance messages, you must also specify at least one of the following options:
  - **-qsmp**
  - **-O5**
  - **-qipa=level=2**
- When used with an option that produces data reorganizations such as **-qipa=level=2**, the report contains information about those reorganizations.

If no *contentSelectionList* options are selected in their positive form, all available report information is produced.

Predefined macros

None.

Examples

If you want to compile myprogram.c to produce an XML report that shows how loops are optimized, enter:

```
bgxlc -qhot -O3 -qlistfmt=xml=transforms myprogram.c
```

If you want to compile myprogram.c to produce an XML report that shows which functions are inlined, enter:

```
bgxlc -qinline -qlistfmt=xml=inlines myprogram.c
```

genhtml command

To view the HTML version of an XML report that has already been generated, you can use the **genhtml** tool.

Use the following command to view the existing XML report in HTML format. This command generates the HTML content to standard output.

```
genhtml xml_file
```

Use the following command to generate the HTML content into a defined HTML file. You can use a web browser to view the generated HTML file.

```
genhtml xml_file > target_html_file
```

Note: The extension of the HTML file name must be compliant with the static HTML page standard, for example, `.html` or `.htm`. Otherwise, the web browser might not be able to open the file.

Related information

- "-qreport" on page 230
- "Using compiler reports to diagnose optimization opportunities" in the *XL C/C++ Optimization and Programming Guide*

-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 17.

Syntax

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:

```
bgxlc myprogram.c -qlistopt
```

Related information

- "-qlist" on page 191
- "-qprint" on page 223
- "-qsource" on page 251

-qlonglit

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

In 64-bit mode, when determining the implicit types for integer literals, the compiler behaves as if an `l` or `L` suffix were added to integral literals with no suffix or with a suffix consisting only of `u` or `U`.

Syntax

Defaults

-qnolonglit

Usage

After you specify the **-qlonglit** option, if the `int` or `unsigned int` type is contained in the implicit type list of a integer literal, the `int` or `unsigned int` type is replaced with the `long int` or `unsigned long int` type, respectively. For more information about the integer literals, see "Integer literals".

Predefined macros

None.

Examples

In 64-bit mode, after you specify the **-qlonglit** option, the integer literal 0x80000000 has the `long int` type. Otherwise, if this option is not specified, the integer literal has the `unsigned int` type.

-qlonglong

Category

Language element control

Pragma equivalent

#pragma options [no]longlong

Purpose

Allows IBM `long long` integer types in your program.

Syntax

Defaults

- C **-qlonglong** for the bgxlc, bgxlc++, bgxlC, bgcc and bgc99 invocation commands; **-qnolonglong** for the bgc89 invocation command.
- C++ **-qlonglong**

Usage

C This option takes effect when the **-qlanglvl=extended | stdc89 | extc89** option is in effect. It is not valid when the **-qlanglvl=stdc99 | extc99** option is in effect, because the `long long` support provided by this option is incompatible with the semantics of the `long long` types mandated by the C99 standard.

C++ This option does not take effect when the **-qlanglvl=c99longlong** option is in effect, because the `long long` support provided by this option is incompatible with the semantics of the `long long` types mandated by the C99 standard as adopted in C++0x.

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:

```
bgcc myprogram.c -qlonglong
```

Related information

- "Integral types" in the *IBM XL C/C++ for Blue Gene/Q, V12.1 Language Reference*

-ma (C only)

See “-qalloca, -ma (C only)” on page 68.

-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 11. 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:

```
bgxlc -c -qmakedep mysource.c
```

To compile `foo_src.c` and create an output file named `mysource.d`, enter:

```
bgxlc -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:

```
bgxlc -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:

```
bgxlc -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:

```
bgxlc -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:

```
bgxlc -c -qmakedep foo_src1.c foo_src2.c -MF c:/tmp/
```

Related information

- "-MF" on page 203
- "-o" on page 206
- "Directory search sequence for include files" on page 11

-qmaxerr

Category

Error checking and debugging

Pragma equivalent

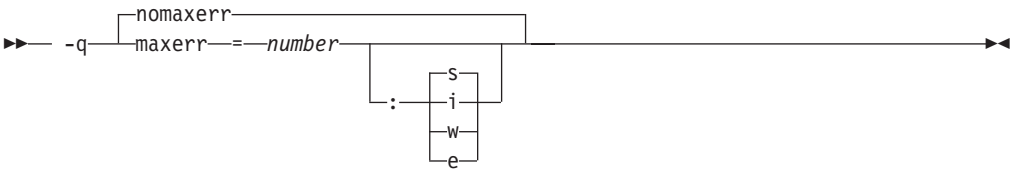
None.

Purpose

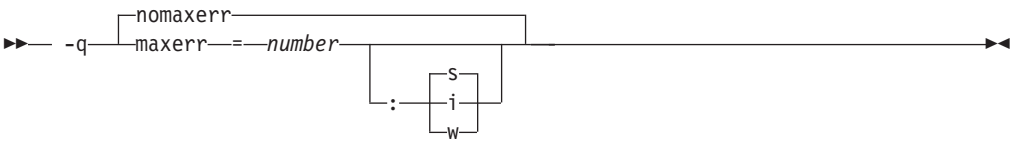
Stops compilation when the number of error messages of a specified severity level or higher reaches a specified number.

Syntax

-qmaxerr syntax — C



-qmaxerr syntax — C++



Defaults

-qnomaxerr

Parameters

number
It specifies the maximum number of messages the compiler generates before it stops. *number* must be an integer with a value of 1 or greater.

i Specifies that the severity level is Informational (I) or higher.

w Specifies that the severity level is Warning (W) or higher.

C **e**
Specifies that the severity level is Error (E) or higher.

s Specifies that the severity level is Severe (S).

Usage

If the **-qmaxerr** option does not specify the severity level, it uses the severity that is in effect by the **-qhalt** option; otherwise, the severity level is specified by either **-qmaxerr** or **-qhalt** that appears last.

Diagnostic messages can be controlled by the **-qflag** option.

Predefined macros

None.

Examples

To stop compilation of `myprogram.c` when 10 warnings are encountered, enter the command:

```
bgxlc 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:

```
bgxlc myprogram.c -qmaxerr=5
```

To stop compilation of `myprogram.c` when 3 informational messages are encountered, enter the command:

```
bgxlc myprogram.c -qmaxerr=3:i
```

or:

```
bgxlc myprogram.c -qmaxerr=3 -qhalt=i
```

Related information

- "-qflag" on page 107
- "-qhalt" on page 127
- "Message severity levels and compiler response" on page 15

-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

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:

```
bgxlc 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 **nodbcs** is in effect, the compiler treats all literals as single-byte literals.

Syntax

Defaults

-qnombcs, -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:

  ```
  bgxlc /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.lst

Predefined macros

None.

Examples

To compile myprogram.c if it contains multibyte characters, enter:

```
bgxlc myprogram.c -qmbcs
```

Related information

- "-D" on page 93

-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 198 for more information.

Syntax

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 198
- "-o" on page 206
- "Directory search sequence for include files" on page 11

-qminimaltoc

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; 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

Defaults

-qnominimaltoc

Usage

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

Category

Output control

Pragma equivalent

None.

Purpose

Creates a shared object from generated object files.

Use this option, together with the related options described later in this topic, 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

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 **-qmkshrobj**:

-o *shared_file*
The name of the file that holds the shared file information. The default is a.out.

-e *name*
Sets the entry name for the shared executable to *name*.

Note: Options **-qmkshrobj** and **-qstaticlink** are incompatible and cannot be specified together. If you specify **-qmkshrobj** and **-qstaticlink** (or **-qstaticlink=libgcc**) together, the driver issues the following message: `1501–264 The options $1 and $2 are incompatible. Option $1 is ignored.`

For detailed information about 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, enter the following command:

```
bgxlc -qmkshrobj -o big_lib.so lib_a.o lib_b.o lib_c.o
```

Related information

- "-e" on page 99
- "-qipa" on page 151
- "-o"
- "-qpic" on page 220
- "-qpriority (C++ only)" on page 224

-o

Category

Output control

Pragma equivalent

None.

Purpose

Specifies a name for the output object, assembler, or executable file.

Syntax

Defaults

See "Types of output files" on page 4 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:

```
bgxlc myprogram.c -o myaccount
```

To compile `test.c` to an object file only and name the object file `new.o`, enter:

```
bgxlc test.c -c -o new.o
```

Related information

- "-c" on page 77
- "-E" on page 100
- "-P" on page 215
- "-qsyntaxonly (C only)" on page 269

-O, -qoptimize

Category

Optimization and tuning

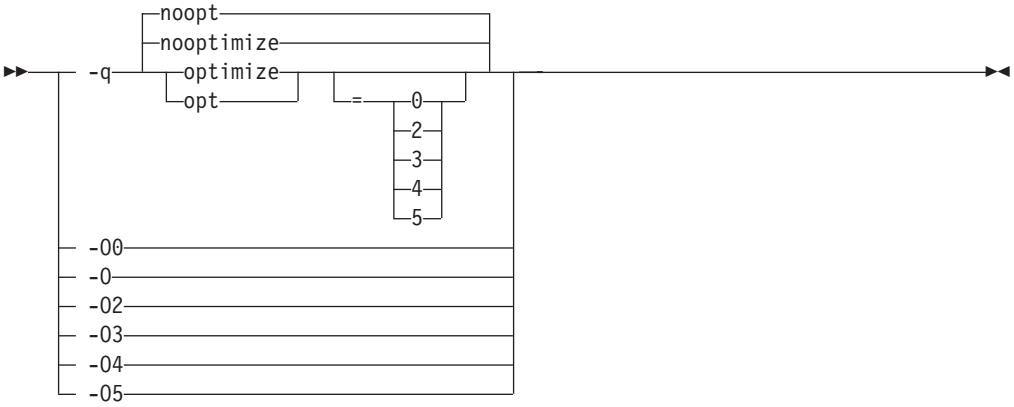
Pragma equivalent

#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[60000000003]`, 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. Specifying **-O3** implies **-qhot=level=0**, unless you explicitly specify **-qhot** or **-qhot=level=1** option.

-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" on page 113 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 **-qarch** and **-qtune** options to the architecture of the compiling machine
- Sets the **-qcache** option most appropriate to the characteristics of the compiling machine
- 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.

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:

```
bgxlc myprogram.c -O3
```

Related information

- "-qhot" on page 130
- "-qipa" on page 151
- "-qstrict" on page 261
- "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

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**, **-qsmp,** or **-qipa**. 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.

The naming rules of a .optdbg file are as follows:

- If a .optdbg file is generated at the compile step, its name is based on the output file name of the compile step.
- If a .optdbg file is generated at the link step, its name is based on the output file name of the link step.

If you compile and link in the same step using the **-qoptdebug** option with **-qipa**, the .optdbg file is generated only at the link step.

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 GuideXL C/C++ Optimization and Programming Guide*.

Related information

- "-O, -qoptimize" on page 207
- "-qhot" on page 130
- "-qipa" on page 151
- "-qsmp" on page 247
- "-g" on page 121
- "-qlinedebug" on page 190

-qoptfile

Category

Compiler customization

Pragma equivalent

None.

Purpose

Specifies a file containing a list of additional command line options to be used for the compilation.

Syntax

Defaults

None.

Parameters

filename
Specifies the name of the file that contains a list of additional command line options. *filename* can contain a relative path or absolute path, or it can contain no path. It is a plain text file with one or more command line options per line.

Usage

The format of the option file follows these rules:

- Specify the options you want to include in the file with the same syntax as on the command line. The option file is a whitespace-separated list of options. The following special characters indicate whitespace: `\n`, `\v`, `\t`. (All of these characters have the same effect.)
- A character string between a pair of single or double quotation marks are passed to the compiler as one option.
- You can include comments in the options file. Comment lines start with the # character and continue to the end of the line. The compiler ignores comments and empty lines.

When processed, the compiler removes the **-qoptfile** option from the command line, and sequentially inserts the options included in the file before the other subsequent options that you specify.

The **-qoptfile** option is also valid within an option file. The files that contain another option file are processed in a depth-first manner. The compiler avoids infinite loops by detecting and ignoring cycles in option file inclusion.

If **-qoptfile** and **-qsaveopt** are specified on the same command line, the original command line is used for **-qsaveopt**. A new line for each option file is included representing the contents of each option file. The options contained in the file are saved to the compiled object file.

Predefined macros

None.

Examples

This is an example of specifying an option file.

```
$ cat options.file
# To perform optimization at -O4 level, and high-order
# loop analysis and transformations during optimization
-O4 -qhot
# To generate position-independent code
-qpic

$ bgxlC -qlist -qoptfile=options.file -qipa test.c
```

The preceding example is equivalent to the following invocation:

```
$ bgxlC -qlist -O4 -qhot -qpic -qipa test.c
```

This is an example of specifying an option file that contains -qoptfile with a cycle.

```
$ cat options.file2
# To perform optimization at -O4 level, and high-order
# loop analysis and transformations during optimization
-O4 -qhot
# To include the -qoptfile option in the same option file
-qoptfile=options.file2
# To generate position-independent code
-qpic
# To produce a compiler listing file
-qlist

$ bgxlC -qlist -qoptfile=options.file2 -qipa test.c
```

The preceding example is equivalent to the following invocation:

```
$ bgxlC -qlist -04 -qhot -qpic -qlist -qipa test.c
```

This is an example of specifying an option file that contains -qoptfile without a cycle.

```
$ cat options.file1
-O4 -qhot
-qoptfile=options.file2
-qfixed

$ cat options.file2
-qfree

$ bgxlC -qoptfile=options.file1 test.c
```

The preceding example is equivalent to the following invocation:

```
$ bgxlC -O4 -qhot -qfree -qfixed test.c
```

This is an example of specifying -qsaveopt and -qoptfile on the same command line.

```
$ cat options.file3
-O4
-qhot

$ bgxlC -qsaveopt -qipa -qoptfile=options.file3 test.c -c

$ what test.o
test.o:
opt f bgxlC -qsaveopt -qipa -qoptfile=options.file3 test.c -c
optfile options.file3 -O4 -qhot
```

Related information

- "-qsaveopt" on page 239

-p, -pg, -qprofile

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:

```
bgxlc myprogram.c -p
```

Remember to compile *and* link with one of the profiling options. For example:

```
bgxlc myprogram.c -p -c
bgxlc myprogram.o -p -o program
```

Related information

- "-qtbtable" on page 272
- 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

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 78
- "-E" on page 100
- "-qppline" on page 222
- "-qsyntaxonly (C only)" on page 269

-qpack_semantic

Category

Portability and migration

Pragma equivalent

None.

Purpose

Controls the syntax and semantics of the **#pragma pack** directive.

Syntax

Defaults

-qpack_semantic=ibm

Parameters

gnu
Uses the GCC syntax and semantics for **#pragma pack**. The effects of this suboption are:

- The current packing value is independent from the packing stack.
- Values are only placed on the pack stack with the **push** parameter. Only values that have been specified with the **push** parameter can be removed from the stack by the **pop** parameter.
- If a **#pragma pack** directive is specified inside a nested aggregate, it affects the outer, containing aggregate as well.

ibm
Uses the IBM syntax and semantics for **#pragma pack**. The effects of this suboption are:

- The current packing value is automatically placed at the top of the packing stack.
- There is no **push** parameter. Any value can be removed from the stack by the **pop** parameter.
- If a **#pragma pack** directive is specified inside a nested aggregate, it only affects aggregates that follow it; that is, it can only affected the inner aggregates.

See "#pragma pack" on page 338 for full details on the syntax and semantics, as well as examples, of these suboptions.

Usage

You should not need to use this option unless you are porting applications compiled with GCC and need to preserve source-level compatibility with the GCC version of the pragma directive.

Predefined macros

None.

Examples

See "#pragma pack" on page 338 for examples.

-qpath

Category

Compiler customization

Pragma equivalent

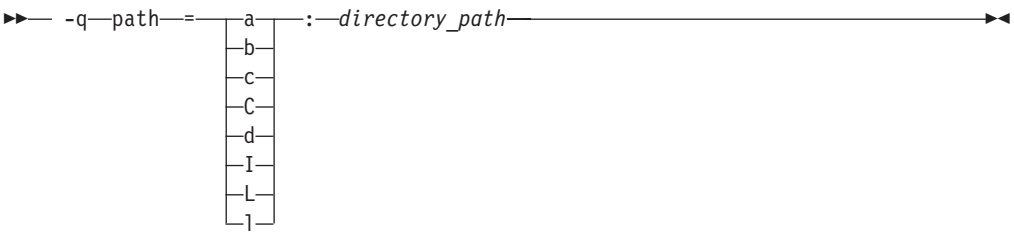
None.

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	as
b	Low-level optimizer	xlCcode
c	Compiler front end	xlcentry, xlCentry
C++ 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	ld

Usage

The **-qpath** option overrides the **-F**, **-t**, and **-B** options.

Predefined macros

None.

Examples

To compile `myprogram.c` using a substitute **bgxlc** compiler in `/lib/tmp/mine/` enter:

```
bgxlc myprogram.c -qpath=c:/lib/tmp/mine/
```

To compile `myprogram.c` using a substitute linker in `/lib/tmp/mine/`, enter:

```
bgxlc myprogram.c -qpath=l:/lib/tmp/mine/
```

Related information

- "-B" on page 75
- "-F" on page 106
- "-t" on page 270

-qphsinfo

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Reports the time taken in each compilation phase to standard output.

Syntax

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 compile time and the time that the CPU spends handling system calls.

The time reported by `-qphsinfo` is in seconds.

Predefined macros

None.

Examples

C To compile `myprogram.c` and report the time taken for each phase of the compilation, enter:

```
bgxlc 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
```

C++ To compile `myprogram.C` and report the time taken for each phase of the compilation, enter:

```
bgxlc++ 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

- **-qpic=small**

Parameters

small
Instructs the compiler to assume that the size of the Table of Contents (TOC) is no larger than 64 Kb. When **-qpic=small** is in effect, the compiler generates one instruction for each GOT or TOC access.

large
Instructs the compiler to assume that the size of the TOC is larger than 64 Kb in 64-bit mode. When **-qpic=large** is in effect, the compiler generates two instructions for each TOC access to enlarge the accessing range. This helps avoid TOC overflow conditions when the Table of Contents is larger than 64 Kb.

Specifying **-qpic** without any suboptions is equivalent to **-qpic=small**.

Usage

When **-q64** is in effect, **-qpic** is always enabled.

When you specify `-qpic=large -qtls -q64`, thread local storage (TLS) symbols are not affected by **-qpic=large**.

You can use different TOC access options for different compilation units in an application.

Note: For applications whose TOC size is larger than 64K, using **-qpic=large** can improve performance. However, for applications whose TOC is smaller than 64K, using **-qpic=large** slows down the program. To decide whether to use **-qpic=large**, compile the program with **-qpic=small** first. If an overflow error message is generated, use **-qpic=large** instead.

Predefined macros

None.

Examples

To compile a shared library `libmylib.so`, use the following commands:

```
bgxlc mylib.c -qpic=small -c -o mylib.o
bgxlc -qmkshrobj mylib -o libmylib.so.1
```

Related information

- "-q64" on page 62
- "-qmkshrobj" on page 205

-qppline

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

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:

```
bgxlc myprogram.c -P -qppline
```

Related information

- "-E" on page 100
- "-P" on page 215

-qprefetch

Category

Optimization and tuning

Pragma equivalent

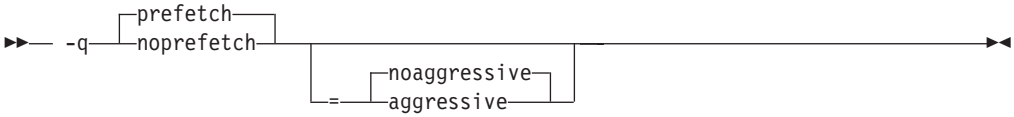
None.

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**
- **-qprefetch=noaggressive**

Parameters

aggressive | noaggressive
This suboption guides the compiler to generate aggressive data prefetching at optimization level **-O3 -qhot** or higher. If you do not specify **aggressive**, **-qprefetch=noaggressive** is implied.

Usage

The **-qnoprefetch** option does not prevent built-in functions such as **__prefetch_by_stream** from generating prefetch instructions.

Related information

- "-qarch" on page 69
- "-qhot" on page 130
- "-qreport" on page 230
- "__mem_delay" on page 489

-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:

```
bgxlc myprogram.c -qnoprint
```

-qpriority (C++ only)

Category

Object code control

Pragma equivalent

#pragma options priority, #pragma priority

Purpose

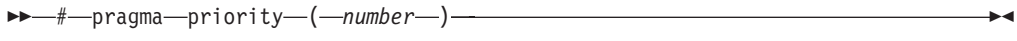
Specifies the priority level for the initialization of static objects.

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

Pragma syntax



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:

```
bgxlc++ myprogram.C -c -qpriority=2000
```

Related information

- "Initializing static objects in libraries" in the *XL C/C++ Optimization and Programming Guide*

-qprocimported, -qproclocal, -qprocunknown

Category

Optimization and tuning

Pragma equivalent

#pragma options proclocal, #pragma options procimported, #pragma options procunknown

Purpose

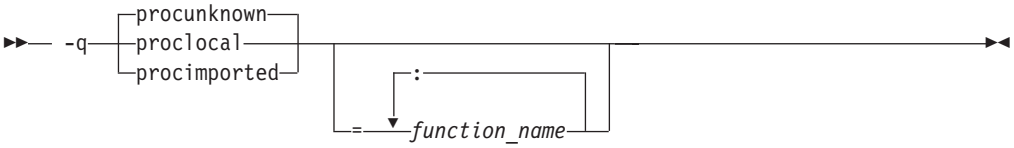
Marks functions as local, imported, or unknown in 64-bit compilations.

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 **procunkown** 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

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:

```
bgxlc 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 **-qproclocal**:

```
int main(void)
{
    printf("Just in function foo1()\n");
    printf("Just in function foo1()\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 `-qproclocal -qprocimported=printf`.

Related information

- "-qdataimported, -qdatalocal, -qtocdata" on page 94

-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

Defaults

-qnoproto

Usage

This option is only valid when the compiler allows unprototyped functions; that is, with the **bgcc** or **bgxlc** 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:

```
bgxlc my_c_program.c -qproto
```

-r

Category

Object code control

Pragma equivalent

None.

Purpose

Produces a nonexecutable output file to use as an input file in another ld command call. This file may also contain unresolved symbols.

Syntax

Defaults

Not applicable.

Usage

A file produced with this flag is expected to be used as an input file in another compiler invocation or ld command call.

Predefined macros

None.

Examples

To compile `myprogram.c` and `myprog2.c` into a single object file `mytest.o`, enter:

```
bgxlc 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

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:

```
bgxlc myprogram.c -lspfiles -R/usr/tmp/old
```

Related information

- "-L" on page 164

-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 automatic parallelization or vectorization, the listing file shows a pseudo-C code listing and a summary of how program loops are parallelized or optimized. The report also includes diagnostic information to show why specific loops could not be parallelized or vectorized. For instance, when **-qreport** is used with **-qsimd=auto**, messages are provided to identify non-stride-one references that can 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.

Syntax

Defaults

-qnoreport

Usage

For **-qreport** to generate a loop transformation listing, you must also specify one of the following on the command line:

- **-qsimd=auto**
- **-qsmp**
- **-qhot=level=2** and **-qsmp**

- **-O5**
- **-qipa=level=2**

For **-qreport** to generate a parallel transformation listing or parallel performance messages, you must also specify one of the following options on the command line:

- **-qsmp**
- **-O5**
- **-qipa=level=2**

To generate data reorganization information, specify the optimization level **-qipa=level=2** or **-O5** together with **-qreport**. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

To generate information about data prefetch insertion locations, use the optimization level of **-qhot**, or any other option that implies **-qhot** together with **-qreport**. This information appears in the `LOOP TRANSFORMATION SECTION` of the listing file.

To generate a list of aggressive loop transformations and parallelizations performed on loop nests in the `LOOP TRANSFORMATION SECTION` of the listing file, use the optimization level of **-qhot=level=2** and **-qsmp** together with **-qreport**.

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:

```
bgxlc -qhot -O3 -qreport myprogram.c
```

To compile `myprogram.c` so the compiler listing also includes a report showing how parallelized loops are transformed, enter:

```
bgxlc_r -qhot -qsmp -qreport myprogram.c
```

Related information

- “-qhot” on page 130
- “-qsimd” on page 243
- “-qipa” on page 151
- “-qsmp” on page 247
- “-qoptdebug” on page 211
- "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

Defaults

Not applicable.

Parameters

register_name
A valid register name on the target platform. Valid registers are:

r0 to r31
General purpose registers

f0 to f31
Floating-point registers

v0 to v31
Vector registers (on selected processors only)

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:

```
bgxlc myprogram.c -qreserved_reg=r3:r4
```

Related information

- "Variables in specified registers" in the *XL C/C++ Language Reference*

-qrestrict (C only)

Category

Optimization and tuning

Pragma equivalent

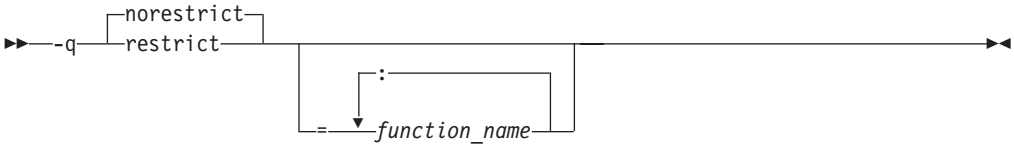
None.

Purpose

Indicates to the compiler that no other pointers can access the same memory that has been addressed by function parameter pointers.

Specifying this option is equivalent to adding the `restrict` keyword to the pointer parameters within the specified functions, except that you do not need to modify the source file.

Syntax



Defaults

-qnorestrict. It means no function pointer parameters are restricted, unless you specify the **restrict** attribute in the source.

Usage

If you do not specify the *function_name*, pointer parameters in all functions are treated as **restrict**. Otherwise, only those pointer parameters in the listed functions are treated as **restrict**.

function_name is a colon-separated list.

Using this option can improve the performance of your application, but incorrectly asserting this pointer restriction might cause the compiler to generate incorrect code based on the false assumption. If the application works correctly when recompiled without **-qrestrict**, the assertion might be false. In this case, this option should not be used.

Notes:

- Using **-qnokeyword=restrict** has no impact on the **-qrestrict** option.
- If you use both **-qalias=norestrict** and the **-qrestrict** option, the last one specified is used.

Predefined macros

None.

Examples

To compile `myprogram.c`, instructing the compiler to restrict the pointer access, enter:

```
bgxlc -qrestrict myprogram.c
```

Related information

- The restrict type qualifier in the *XL C/C++ Language Reference*.
- Keywords in the *XL C/C++ Language Reference*.
- -qkeyword
- -qalias

-qro

Category

Object code control

Pragma equivalent

#pragma options ro, #pragma strings

Purpose

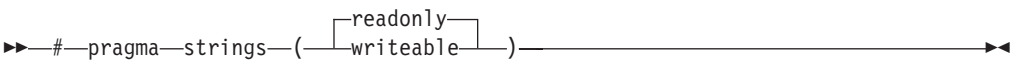
Specifies the storage type for string literals.

When **ro** or **strings=readonly** is in effect, strings are placed in read-only storage. When **noro** or **strings=writeable** is in effect, strings are placed in read/write storage.

Syntax

Option syntax

Pragma syntax



Defaults

C Strings are read-only for all invocation commands except **bgcc**. If the **bgcc** 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:

```
bgxlc myprogram.c -qnoro
```

Related information

- "-qro" on page 234
- "-qroconst"

-qroconst

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 **noroconst** is in effect, constants are placed in read/write storage.

Syntax

Defaults

- C **-qroconst** for all compiler invocations except **bgcc** and its derivatives. **-qnoroconst** for the **bgcc** 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 want to specify storage characteristics of both string literals (**-qro**) and constant values (**-qroconst**).

Predefined macros

None.

Related information

- "-qro" on page 234

-qrtti (C++ 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

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)" on page 101

-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

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 debugging information using options such as **-g**.

Predefined macros

None.

Related information

- "-g" on page 121

-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

Defaults

Not applicable.

Usage

You can invoke the assembler with any compiler invocation command. For example,

```
bgxlc 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:

```
bgxlc myprogram1.c myprogram2.c -o -S
```

Predefined macros

None.

Examples

To compile `myprogram.c` to produce an assembler language file `myprogram.s`, enter:

```
bgxlc myprogram.c -S
```

To assemble this program to produce an object file `myprogram.o`, enter:

```
bgxlc myprogram.s -c
```

To compile `myprogram.c` to produce an assembler language file `asmprogram.s`, enter:

```
bgxlc myprogram.c -S -o asmprogram.s
```

Related information

- "-E" on page 100
- "-P" on page 215

-qsaveopt

Category

Object code control

Pragma equivalent

None.

Purpose

Saves the command-line options used for compiling a source file, the user's configuration file name and the options specified in the configuration files, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.

Syntax

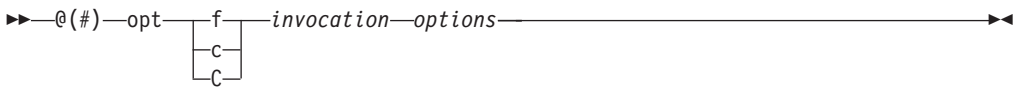
Defaults

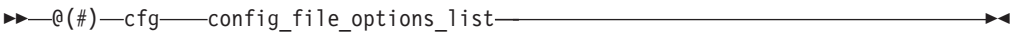
-qnosaveopt

Usage

This option has effect only when compiling to an object (.o) file (that is, using the **-c** option). Though each object might 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:





where:

f Signifies a Fortran language compilation.

c Signifies a C language compilation.

C Signifies a C++ language compilation.

invocation
Shows the command used for the compilation, for example, **bgxlc**.

options The list of command line options specified on the command line, with individual options separated by space.

config_file_options_list
The list of options specified by the **options** attribute in all configuration files that take effect in the compilation, separated by space.

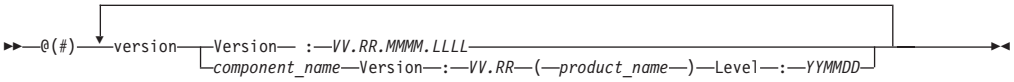
env_var_definition
The environment variables that are used by the compiler. Currently only **XLC_USR_CONFIG** is listed.

Note: You can always use this option, but the corresponding information is only generated when the environment variable **XLC_USR_CONFIG** is set.

For more information about the environment variable **XLC_USR_CONFIG**, see Compile-time and link-time environment variables.

Note: The string of the command-line options is truncated after 64k bytes.

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:



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:

```
bgxlc 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/bg/12.1/bin/.orig/bgxlc
-F/opt/ibmcmp/vac/bg/12.1/etc/vac.cfg.sles10.gcc412_20070115 t.c -qsaveopt -qhot -c
cfg -qlanglvl=extc99 -qcpluscmt -qkeyword=inline -qalias=ansi -qtls
-D_CALL_SYSV -D__null=0 -D__NO_MATH_INLINES
-qnoautoconfig -qdebug=nblrl -q64 -qarch=qp -qtune=qp
-qcache=level=1:type=i:size=16:line=64:assoc=4:cost=6
-qcache=level=1:type=d:size=16:line=64:assoc=8:cost=6
-qcache=level=2:type=c:size=33554:line=128:assoc=16:cost=80 -qstaticlink -qtls -qtls
version IBM XL C/C++ for Blue Gene, V12.0
version Version: 12.00.0000.0002
version Driver Version: 12.00(C/C++) Level: YYMMDD
version C Front End Version: 12.00(C/C++) Level: YYMMDD
version High-Level Optimizer Version: 12.00(C/C++) and 14.00(Fortran) Level: YYMMDD
version Low-Level Optimizer Version: 12.01(C/C++) and 14.01(Fortran) Level: YYMMDD
```

In the first line, `/opt/ibmcmp/vac/bg/12.1/bin/bgxlc` identifies the source used as C, `/opt/ibmcmp/vacpp/bg/12.1/bin/bgxlc` 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 292

-qshowinc

Category

Listings, messages, and compiler information

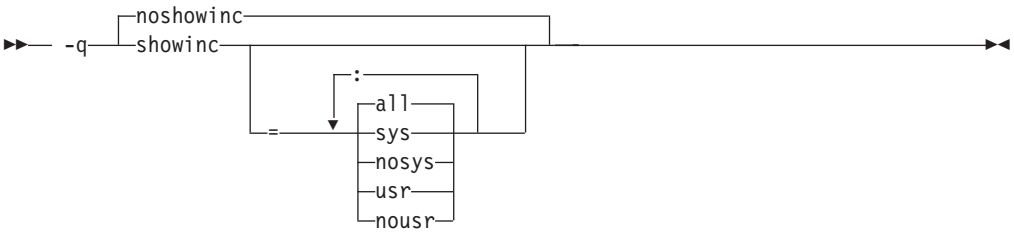
Pragma equivalent

#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:

```
bgxlc myprogram.c -qsource -qshowinc
```

Related information

- "-qsource" on page 251

-qshowmacros

Category

"Output control" on page 43

Pragma equivalent

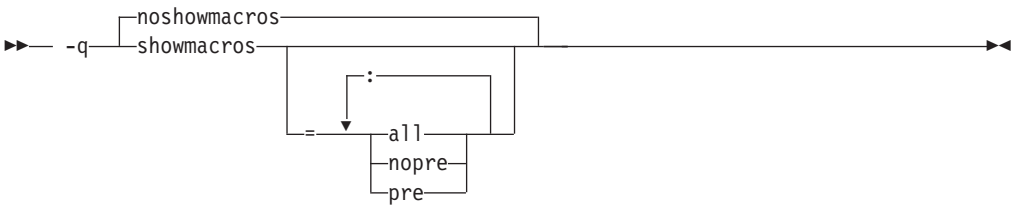
None

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 100
- "-P" on page 215

-qsimd

Category

Optimization and tuning

Pragma equivalent

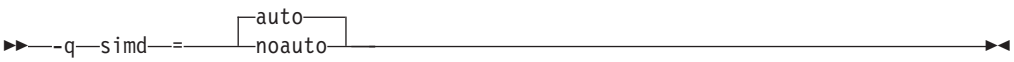
#pragma nosimd

Purpose

Controls whether the compiler can automatically take advantage of vector instructions for processors that support them.

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

Syntax



Defaults

-qsimd=auto

Usage

On Blue Gene/Q platforms, the **-qsimd=auto** option enables automatic generation of QPX vector instructions. It is enabled by default at all optimization levels.

When **-qsimd=auto** is in effect, the compiler converts certain operations that are performed in a loop on successive elements of an array into vector instructions. These instructions calculate several results at one time, which is faster than calculating each result sequentially. Applying this option is useful for applications with significant image processing demands.

The **-qsimd=noauto** option disables the conversion of loop array operations into vector instructions. Finer control can be achieved by using **-qstrict=ieeefp**, **-qstrict=operationprecision**, and **-qstrict=vectorprecision**. For details, see "-qstrict" on page 261.

Note: Using vector instructions to calculate several results at one time might delay or even miss detection of floating-point exceptions on some architectures. If detecting exceptions is important, do not use **-qsimd=auto**.

Rules

The following rules apply when you use the **-qsimd** option:

- Specifying **-qsimd** without any suboption has the same effect as **-qsimd=auto**.
- This option is available only when you set **-qarch** to a target architecture that supports vector instructions.
- If you specify **-qsimd=auto** to enable IPA at the compile time but specify **-qsimd=noauto** at the link time, the compiler automatically sets **-qsimd=auto** and sets an appropriate value for **-qarch** to match the architecture specified at the compile time.

Predefined macros

None.

Example

The following example shows the usage of **#pragma nosimd** to disable **-qsimd=auto** for a specific `for` loop:

```
...
#pragma nosimd
for (i=1; i<1000; i++) {
    /* program code */
}
```

Related information

- "-qarch" on page 69
- "-qstrict" on page 261

-qskipsrc

Category

"Listings, messages, and compiler information" on page 52

Pragma equivalent

None.

Purpose

When a listing file is generated using the **-qsource** option, **-qskipsrc** can be used to determine whether the source statements skipped by the compiler are shown in the source section of the listing file. Alternatively, the **-qskipsrc=hide** option is used to hide the source statements skipped by the compiler.

Syntax

Defaults

- -qskipsrc=show

Parameters

show | hide

When **show** is in effect, the compiler will display all source statements in the listing. This will result in both true and false paths of the preprocessing directives to be shown.

On the contrary, when **hide** is enabled, all source statements that the compiler skipped will be omitted.

Usage

In general, the **-qskipsrc** option does not control whether the source section is included in the listing file, it only does so when the **-qsource** option is in effect.

To display all source statements in the listing (default option):

```
bgxlc myprogram.c -qsource -qskipsrc=show
```

To omit source statements skipped by the compiler:

```
bgxlc myprogram.c -qsource -qskipsrc=hide
```

Predefined macros

None.

Related information

- "-qsource" on page 251
- "-qshowinc" on page 241
- "-qsrcmsg (C only)" on page 254

-qsmallstack

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Reduces the size of the stack frame.

Syntax

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:

```
bgxlc myprogram.c -qipa -qsmallstack
```

Related information

- "-g" on page 121
- "-qipa" on page 151
- "-O, -qoptimize" on page 207

-qsmp

Category

Optimization and tuning

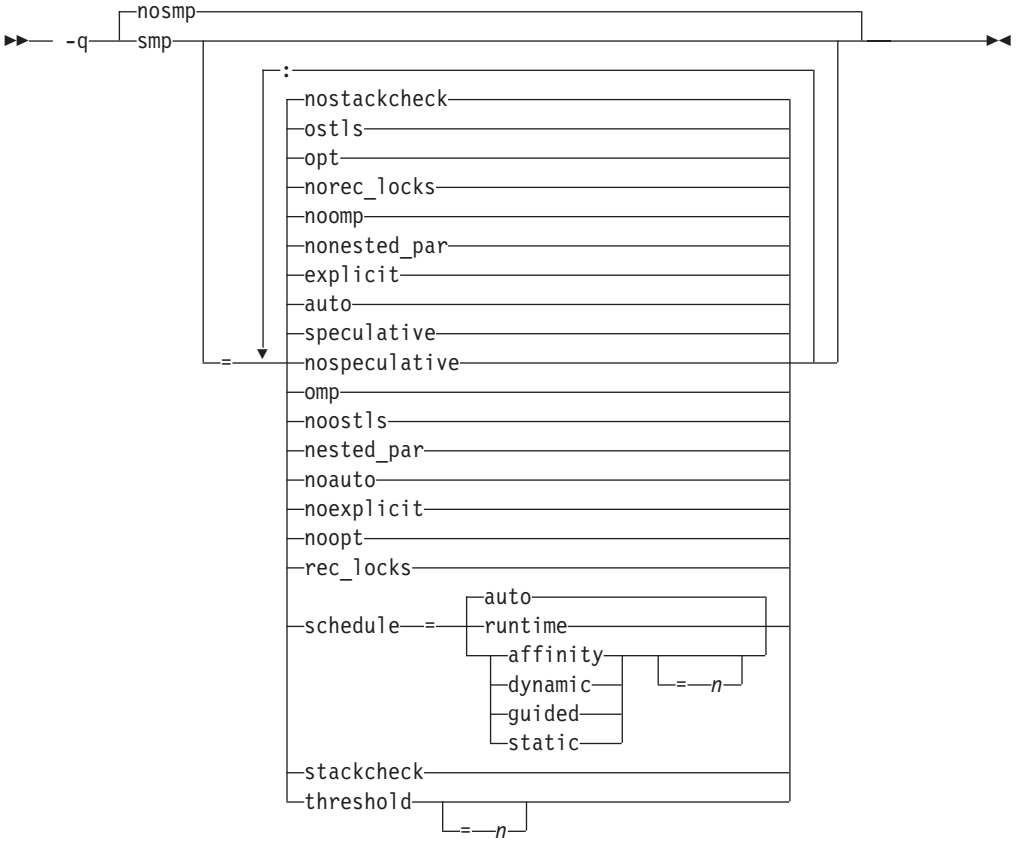
Pragma equivalent

None.

Purpose

Enables parallelization of program code.

Syntax



Defaults

-qnosmp. Code is produced for a uniprocessor machine.

Parameters

auto | noauto
Enables or disables automatic parallelization and optimization of program code. When **noauto** is in effect, only program code explicitly parallelized with OpenMP directives is optimized. **noauto** is implied if you specify **-qsmp=omp** or **-qsmp=noopt**.

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 a 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:

- 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 with the OpenMP standard. When **noomp** is in effect, **auto** is implied. 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.

ostls| noostls
Enables Thread Local Storage (TLS) provided by the operating system to be used for **threadprivate** data. You can use the **noostls** suboption to enable the non-TLS for **threadprivate**. The **noostls** suboption is provided for compatibility with earlier versions.

Note: If you want to use this suboption, your operating system must support TLS to implement OpenMP **threadprivate** data. Use **noostls** to disable OS level TLS if your operating system does not support it.

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:

affinity[=*n*]
The iterations of a loop are initially divided into *n* partitions, containing **ceiling**(*number_of_iterations*/*number_of_threads*) iterations. Each partition is initially assigned to a thread and is then further subdivided into chunks that each contain *n* iterations. If *n* is not specified, then the chunks consist of **ceiling**(*number_of_iterations_left_in_partition* / 2) loop iterations.

When a thread becomes free, it takes the next chunk from its initially assigned partition. If there are no more chunks in that partition, then the thread takes the next available chunk from a partition initially assigned to another thread.

The work in a partition initially assigned to a sleeping thread will be completed by threads that are active.

The **affinity** scheduling type does not appear in the OpenMP API standard.

auto
Scheduling of the loop iterations is delegated to the compiler and runtime systems. The compiler and runtime system can choose any possible mapping of iterations to threads (including all possible valid schedule types) and these might be different in different loops. Do not specify chunk size (*n*).

dynamic[=*n*]
The iterations of a loop are divided into chunks containing *n* iterations each. If *n* is not specified, then the chunks consist of **ceiling**(*number_of_iterations*/*number_of_threads*). iterations.

Active threads are assigned these chunks on a "first-come, first-do" basis. Chunks of the remaining work are assigned to available threads until all work has been assigned.

If a thread is asleep, its assigned work will be taken over by an active thread once that thread becomes available.

guided[=*n*]
The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of *n* loop iterations is reached. If *n* is not specified, the default value for *n* is 1 iteration.

Active threads are assigned chunks on a "first-come, first-do" basis. The first chunk contains **ceiling**(*number_of_iterations*/*number_of_threads*) iterations. Subsequent chunks consist of **ceiling**(*number_of_iterations_left* / *number_of_threads*) iterations.

runtime
Specifies that the chunking algorithm will be determined at run time.

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=auto**.

speculative | nospeculative
Enables thread-level speculative execution.

Note: Specifying **-qsmp=omp** turns off the **speculative** suboption. You need to specify **-qsmp=omp:speculative** to have both OpenMP and thread-level speculative execution.

stackcheck | nostackcheck
Causes the compiler to check for stack overflow by slave threads at run time, and issue a warning if the remaining stack size is less than the number of bytes specified by the **stackcheck** option of the XLSMPOPTS environment variable. This suboption is intended for debugging purposes, and only takes effect when **XLSMPOPTS=stackcheck** is also set; see “XLSMPOPTS” on page 23.

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=auto:explicit:opt:noomp:norec_locks:nonested_par:schedule=auto:
nostackcheck:threshold=100:ostls:speculative
```

Usage

- Specifying the **omp** suboption always implies **noauto**. Specify **-qsmp=omp:auto** to apply automatic parallelization on OpenMP-compliant applications, as well.

- You should only use **-qsmp** with the _**r**-suffixed invocation commands, to automatically link in all of the threadsafe components. You can use the **-qsmp** option with the non-_**r**-suffixed invocation commands, but you are responsible for linking in the appropriate components. . 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 **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 207
- "-qthreaded" on page 278
- "Environment variables for parallel processing" on page 23
- "Pragma directives for parallel processing" on page 355
- "Built-in functions for parallel processing" on page 491

-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 17.

Syntax

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:

```
bgxlc myprogram.c -qsource
```

Related information

- "-qlist" on page 191
- "-qlistopt" on page 195
- "-qprint" on page 223

-qsourcetype

Category

Input control

Pragma equivalent

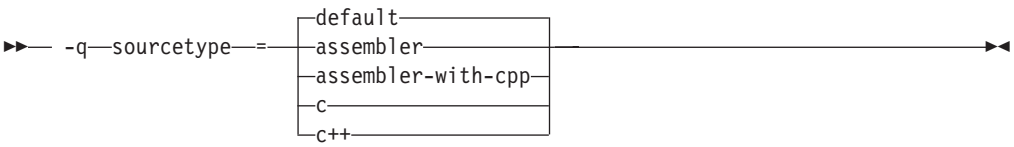
None.

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++ **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:

```
bgxlc 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:

```
bgxlc -qsourcetype=c hello.C
```

Related information

- "-+ (plus sign) (C++ only)" on page 60

-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

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:

```
bgxlc 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

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:

```
bgxlc myprogram.c -qsrcmsg
```

-qstackprotect

Category

"Object code control" on page 48

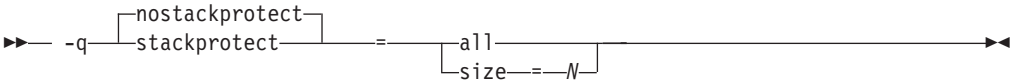
Pragma equivalent

None.

Purpose

Provides protection against malicious code or programming errors that overwrite or corrupt the stack.

Syntax



Defaults

- -qnostackprotect

Parameters

all

all protects all procedures whether or not there are vulnerable objects. This option is not set by default.

size=*N*

size=*N* protects all procedures containing automatic objects greater or equal to *N* bytes in size. The default size is 8 when **-qstackprotect** is enabled.

Note: When both **all** and **size** are used, the last option wins.

Usage

-qstackprotect generates extra code to protect procedures with vulnerable objects against stack corruption. This option is disabled by default because it can cause performance degradation. The default option is **-qnostackprotect**.

To generate code to protect all procedures with vulnerable objects:

```
bgxlc myprogram.c -qstackprotect=all
```

To generate code to protect procedures with objects of certain bytes:

```
bgxlc myprogram.c -qstackprotect=size=8
```

Note:

- This option cannot be used with #pragma options.
- Because of the dependency on **glibc** in Blue Gene/Q, this option requires the following Linux levels:
  - Blue Gene/Q OS with **GLIBC** 2.4 and up (Machines with **GCC** 4.*X* and up).
- All supported Blue Gene/Q systems already support this feature.

Predefined macros

None.

Related information

- "-qinfo" on page 139

-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

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*

-qstaticlink

Category

Linking

Pragma equivalent

None.

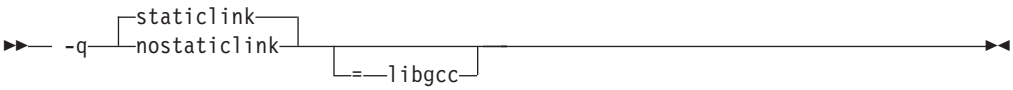
Purpose

Controls how shared and nonshared runtime libraries are linked into an application.

When **-qstaticlink** is in effect, the compiler links only static libraries with the object file being produced. When **-qnostaticlink** is in effect, the compiler links shared libraries with the object file being produced.

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 **libgcc** is specified together with **nostaticlink**, the compiler links to the shared version of **libgcc**. When **libgcc** is specified together with **staticlink**, the compiler links to the static version of **libgcc**.

Usage

Important: 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 that you compile. It is strongly recommended that you seek legal advice before using this option.

The following table shows the equivalent GNU and XL C/C++ options for specifying linkage of shared and nonshared libraries.

Table 27. Option mappings: control of the GNU linker

GNU option	Meaning	XL C/C++ option
-shared	Build a shared object.	-qmkshrobj 1
-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 or -qnostaticlink=libgcc (these two are identical) 2 3
-static-libgcc	Link with the static version of libgcc. You can still link your shared libraries.	-qstaticlink=libgcc 4

Table 27. Option mappings: control of the GNU linker (continued)

GNU option	Meaning	XL C/C++ option
Notes: 1 Options **-qmkshrobj** and **-qstaticlink** are incompatible and cannot be specified together. 2 This is the default setting on SUSE Linux Enterprise Server (SLES) and Red Hat Enterprise Linux (RHEL). 3 On Blue Gene/Q, **-qmkshrobj** implies **-qnostaticlink**. 4 This is the default setting on Blue Gene/Q.		

Predefined macros

None.

Related information

- "-qmkshrobj" on page 205

-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

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:

```
bgxlc 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:

- C The directory specified in the configuration file for the XL C header files (this is normally /opt/ibmcmp/vacpp/bg/12.1/include/) or by the **-qc_stdinc** option
- C++ The directory specified in the configuration file for the XL C and C++ header files (this is normally /opt/ibmcmp/vacpp/bg/12.1/include/) or by the **-qcpp_stdinc** option
- 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

Defaults

-qstdinc

Usage

The search order of header files is described in "Directory search sequence for include files" on page 11.

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:

```
bgxlc myprogram.c -qnostdinc -I/tmp/myfiles
```

Related information

- "-qc_stdinc (C only)" on page 90
- "-qcpp_stdinc (C++ only)" on page 91
- "-qgcc_c_stdinc (C only)" on page 124
- "-qgcc_cpp_stdinc (C++ only)" on page 125
- "-I" on page 133
- "Directory search sequence for include files" on page 11

-qstrict

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]strict

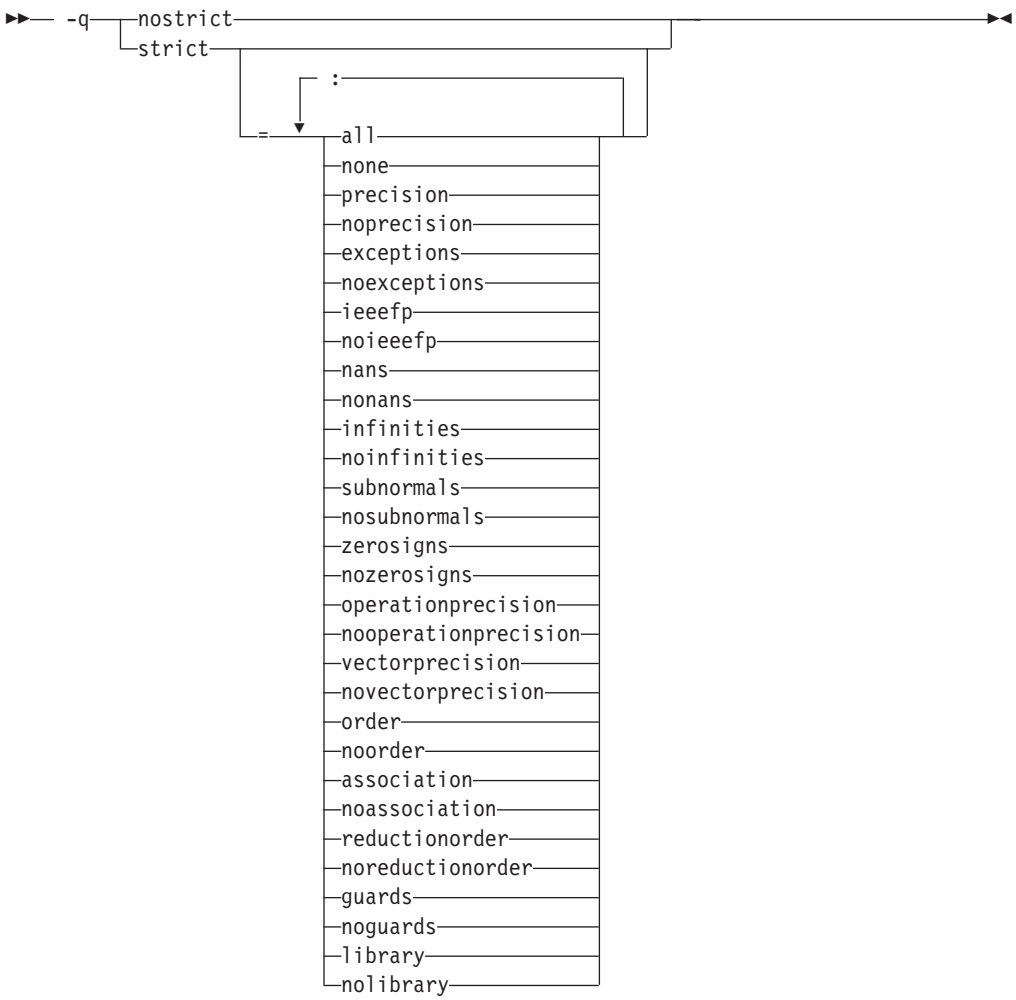
#pragma option_override (*function_name*, "opt (*suboption_list*)")

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 **-qnoopt** 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**, **vectorprecision**, **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.

ieeefp | noieeefp
ieeefp disables transformations that affect IEEE floating-point compliance, including those controlled by the **nans**, **infinities**, **subnormals**, **zerosigns**, **vectorprecision**, and **operationprecision** suboptions. **noieeefp** 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.

vectorprecision | novectorprecision
vectorprecision disables vectorization in loops where it might produce different results in vectorized iterations than in nonvectorized residue iterations. **vectorprecision** ensures that every loop iteration of identical floating-point operations on identical data produces identical results.

novectorprecision enables vectorization even when different iterations might produce different results from the same inputs.

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**, out of loops, or

past function calls that might end the program or throw an exception) which control whether the operation should be executed. **noguards** enables moving operations past guards.

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**, **ieeefp**, 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 the following suboptions:
- **-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**.

Note: For details about the relationship between **-qstrict** suboptions and their **-qfloat** counterparts, see "-qfloat" on page 109.

To override any of these settings, specify the appropriate **-qfloat** suboptions after the **-qstrict** option 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:

```
bgxlc myprogram.c -O3 -qstrict -qfloat=rsqrt
```

To enable all transformations except those affecting precision, specify:

```
bgxlc myprogram.c -qstrict=none:precision
```

To disable all transformations except those involving NaNs and infinities, specify:

```
bgxlc myprogram.c -qstrict=all:nonans:noinfinities
```

Related information

- "-qsimd" on page 243
- "-qfloat" on page 109
- "-qhot" on page 130
- "-O, -qoptimize" on page 207

-qstrict_induction

Category

Optimization and tuning

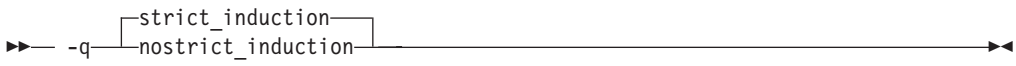
Pragma equivalent

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 207

-qsuppress

Category

Listings, messages, and compiler information

Pragma equivalent

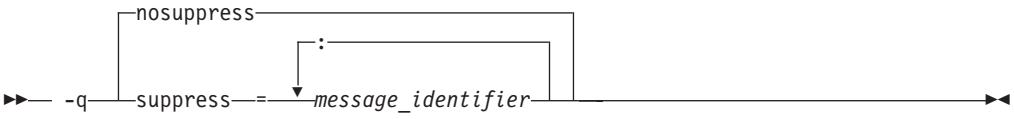
"#pragma report (C++ only)" on page 345

Purpose

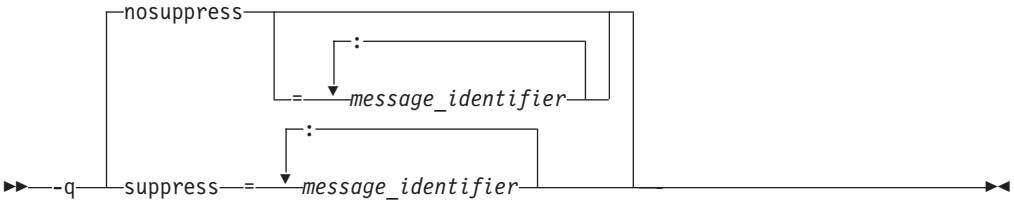
Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.

Syntax

-qsuppress syntax — C



-qsuppress syntax — C++



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:

15 Is the compiler product identifier.

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 codes.

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 **-qhaltonmsg** compiler option has precedence over **-qsuppress**. If both **-qhaltonmsg** and **-qsuppress** are specified, messages that are suppressed by **-qsuppress** are also printed.

C The **-qnosuppress** compiler option cancels previous settings of **-qsuppress**. C

C++ When you specify **-qnosuppress** with specific message identifiers, the previous **-qsuppress** instances with the same message identifiers lose effect. When you specify **-qnosuppress** without specific message identifiers, all previous **-qsuppress** instances lose effect.

If you specify two or three of the following options, the last option specified has precedence:

-qsuppress=*message_identifier*

-qnosuppress=*message_identifier*

-qnosuppress

C++

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:

```
bgxlc myprogram.c -qsuppress=1506-224
```

Related information

- "-qflag" on page 107
- "-qhaltonmsg" on page 128

-qsymtab (C only)

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Determines the information that appears in the symbol table.

Syntax

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:

```
bgxlc myprogram.c -qsymtab=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:

```
bgxlc myprogram.c -g -qsymtab=unref
```

Related information

- "-g" on page 121
- "-qdbxextra (C only)" on page 95
- "-qstatsym" on page 259

-qsyntaxonly (C only)

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Performs syntax checking without generating an object file.

Syntax

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:

```
bgxlc myprogram.c -qsyntaxonly
```

Related information

- "-C, -C!" on page 78
- "-c" on page 77
- "-E" on page 100
- "-o" on page 206
- "-P" on page 215

-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	as
b	Low-level optimizer	xlCcode
c	Compiler front end	xlcentry, xlCentry
C++ 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	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.

Note: If you use the **p** suboption, it can cause the source code to be preprocessed separately before compilation, which can change the way a program is compiled.

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:

```
bgxlc myprogram.c -B/u/newones/compilers/ -tca
```

Related information

- "-B" on page 75

-qtabsize

Category

Language element control

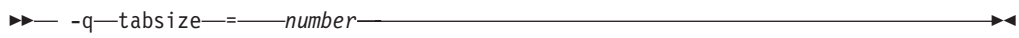
Pragma equivalent

#pragma options tabsize

Purpose

Sets the default tab length, for the purposes of reporting the column number in error messages.

Syntax



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:

```
bgxlc 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

Category

Object code control

Pragma equivalent

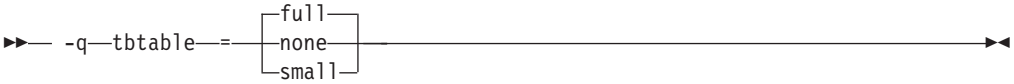
#pragma options tbtable

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



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

The **#pragma** options directive must be specified before the first statement in the compilation unit.

Predefined macros

None.

Related information

- "-g" on page 121

-qtempinc (C++ only)

Category

Template control

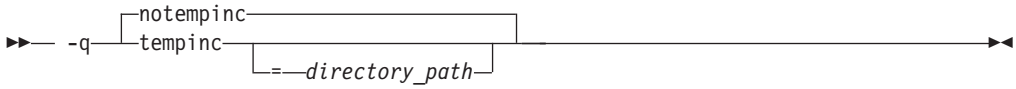
Pragma equivalent

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:

```
bgxlc++ myprogram.C -qtempinc=/tmp/mytemplates
```

Related information

- "#pragma implementation (C++ only)" on page 326
- "-qtmplinst (C++ only)" on page 282
- "-qtemplateregistry (C++ only)" on page 276
- "-qtemplaterecompile (C++ only)" on page 275
- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qtemplatedepth (C++ only)

Category

Template control

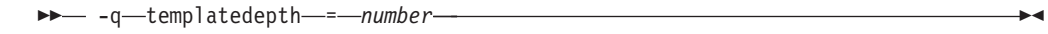
Pragma equivalent

None.

Purpose

Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.

Syntax



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:

```
bgxlc++ 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

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 282
- "-qtempinc (C++ only)" on page 273
- "-qtemplateregistry (C++ only)"
- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qtemplateregistry (C++ only)

Category

Template control

Pragma equivalent

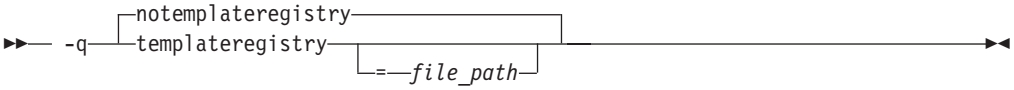
None.

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



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 **-qnotemplateregistry** 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:

```
bgxlc++ myprogram.C -qtemplateregistry=/tmp/mytemplateregistry
```

Related information

- "-qtmplinst (C++ only)" on page 282
- "-qtempinc (C++ only)" on page 273
- "-qtemplaterecompile (C++ only)" on page 275
- "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

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 273
- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qthreaded

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

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.

Related information

- "-qsmp" on page 247

-qtimestamps

Category

"Output control" on page 43

Pragma equivalent

none.

Purpose

Controls whether or not implicit time stamps are inserted into an object file.

Syntax

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

Category

Object code control

Pragma equivalent

None.

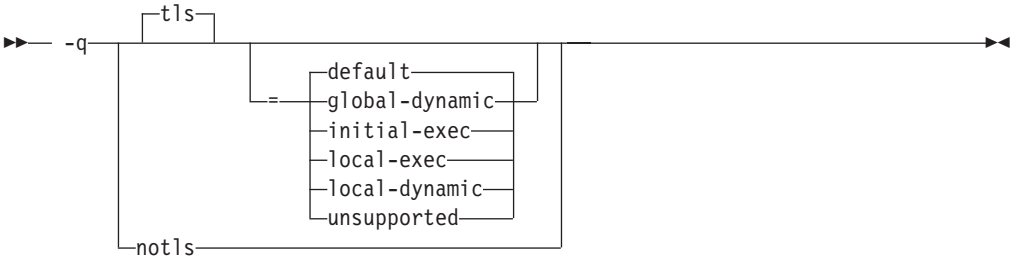
Purpose

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** (**-qpic** 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
- "-qpic" on page 220
- "The __thread storage class specifier" in the *XL C/C++ Language Reference*

-qtm

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables support for transactional memory.

Syntax

Defaults

-qnotm

Usage

The **-qtm** option requires the thread safe compilation mode. Use **-qtm** with a thread safe compiler invocation command such as **bgxlc_r**, **bgxlC_r**, or **bgxlc++_r**.

Related information
- Transactional memory
- #pragma tm_atomic
- Built-in functions for transactional memory

- Environment variables for transactional memory

-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 317.

Predefined macros

None.

Related information

- “-qtemplateregistry (C++ only)” on page 276
- “-qtempinc (C++ only)” on page 273
- “#pragma do_not_instantiate (C++ only)” on page 317
- "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

See “-qdataimported, -qdatalocal, -qtocdata” on page 94.

-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 96
- “-qlanglvl” on page 165

-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

Defaults

The **-qtune=qp** setting is the default setting for **-qarch=qp**, or when no **-qarch** or **-qtune** settings are specified and the **bg**-prefixed commands are used.

Parameters

qp Optimizations are tuned for the Blue Gene/Q platform.

auto
Optimizations are tuned for the platform on which the application is compiled.

Usage

If you specify the following options on the Blue Gene/Q platform, the **-qtune** option is also set:

- Specifying **-q64** also sets **-qtune=qp**.
- Specifying **-O4** or **-O5** also sets **-qtune=auto**.

Predefined macros

None.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to be optimized for a Blue Gene/Q hardware platform, enter:

```
bgxlc -o testing myprogram.c -qtune=qp
```

Related information

- "-qarch" on page 69
- "-q64" on page 62
- "Specifying compiler options for architecture-specific compilation" on page 9
- "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

Defaults

Many macros are predefined by the compiler; see Chapter 5, "Compiler predefined macros," on page 381 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 **-U***name* option has a higher precedence than the **-D***name* 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:

```
bgxlc myprogram.c  -U__unix
```

Related information

- "-D" on page 93

-qunroll

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]unroll, #pragma unroll

Purpose

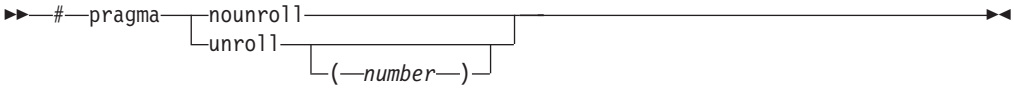
Controls loop unrolling, for improved performance.

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**, **-qsmp**, 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 310
- "#pragma loopid" on page 328
- "#pragma stream_unroll" on page 349
- "#pragma unrollandfuse" on page 350

-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

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)" on page 101

-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. The promotion rule does not apply to types that are larger than `int`.

Syntax

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:

```
bgxlc 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

- "Usual arithmetic conversions" in the *XL C/C++ Language Reference*
- "-qlanglvl" on page 165

-qutf

Category

Language element control

Pragma equivalent

None.

Purpose

Enables recognition of UTF literal syntax.

Syntax

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

Defaults

The compiler does not display the progress of the compilation.

Usage

The **-v** and **-V** options are overridden by the **-#** option.

Predefined macros

None.

Examples

To compile `myprogram.c` so you can watch the progress of the compilation and see messages that describe the progress of the compilation, the programs being invoked, and the options being specified, enter:

```
bgxlc myprogram.c -v
```

Related information

- "-# (pound sign)" on page 61

-qversion

Category

Listings, messages, and compiler information

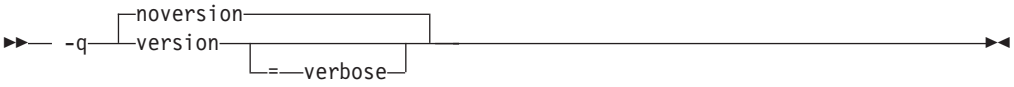
Pragma equivalent

None.

Purpose

Displays the version and release of the compiler being invoked.

Syntax



Defaults

-qnoversion

Parameters

verbose
Additionally displays information about the version, release, and level of each compiler component installed.

Usage

When you specify **-qversion**, the compiler displays the version information and exits; compilation is stopped. If you want to save this information to the output object file, you can do so with the **-qsaveopt -c** options.

-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.

For more details, see Example 1.

-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.

For more details, see Example 2.

Predefined macros

None.

Examples

Example 1:

```
IBM XL C/C++ for Blue Gene/Q, V12.1
Version: 12.01.0000.0001
```

Example 2:

```
IBM XL C/C++ for Blue Gene/Q, V12.1
Version: 12.01.0000.0001
Driver Version: 12.01(C/C++) Level: 060414
C Front End Version: 12.01(C/C++) Level: 060419
C++ Front End Version: 12.01(C/C++) Level: 060420
High Level Optimizer Version: 12.01(C/C++) and 14.01(Fortran) Level: 060411
Low Level Optimizer Version: 12.01(C/C++) and 14.01(Fortran) Level: 060418
```

Related information

- "-qsaveopt" on page 239

-w

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Suppresses informational, language-level and warning messages.

C This option is equivalent to specifying **-qflag=e : e**. C++ This option is equivalent to specifying **-qflag=s : s**.

Syntax

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:

```
bgxlc 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, int)".
```

Related information

- "-qflag" on page 107
- "-qsuppress" on page 266

-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	as
b	Low-level optimizer	xlCcode
c	Compiler front end	xlcentry, xlCentry
C++ C	C++ compiler front end	xlCentry
d	Disassembler	dis
I	High-level optimizer, compile step	ipa
L	High-level optimizer, link step	ipa

Parameter	Description	Executable name
l	Linker	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 **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:

```
bgxlc -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:.

```
bgxlc -Wa,-x -Wl,-s produces_warnings.s uses_many_symbols.c
```

Related information

- "Invoking the compiler" on page 1

-qwarn0x (C++0x)

Note: C++0x is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++0x standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++0x standard and therefore they should not be relied on as a stable programming interface.

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Controls whether to inform users with messages about differences in their programs caused by migration from the C++98 standard to the C++0x standard.

For example, when **-qlanglvl=noc99preprocessor** and **-qwarn0x** are specified, the C++0x preprocessor evaluates the controlling expressions in the `#if` and `#elif` conditional inclusion directives, and compare the evaluation results against that of the non-C++0x preprocessor. If they are different, the compiler issues the following warning message:

```
The preprocessor controlling expression evaluates differently between C++0x
and non-C++0x langlvls.
```

For another example, when you specify both the **-qlanglvl=noc99longlong** and **-qwarn0x** options, the compiler might display messages to indicate that the types of an integer literal are different between the non-C++0x and C++0x language levels.

The following C++0x keywords are not reserved in non-C++0x mode:

- `constexpr`
- `decltype`
- `static_assert`

For each occurrence of these keywords, the compiler issues a message if the corresponding C++0x features and keywords are disabled and if the **-qwarn0x** option is enabled. For example, when you specify both the **-qlanglvl=nostatic_assert** and **-qwarn0x** options, the compiler emits the following message for each `static_assert` token it encounters:

```
C++0x will reserve "static_assert" as a keyword whose C++0x feature can
be enabled by -qlanglvl=static_assert.
```

Syntax

Defaults

-qnowarn0x

Usage

This option is in effect when **-qwarn0x** is set.

Predefined macros

None.

Related information

- "-qlanglvl" on page 165

-qwarn64

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

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

- "-q64" on page 62
- "Compiler messages" on page 14

-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

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:

```
bgxlc 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 17.

Syntax

Defaults

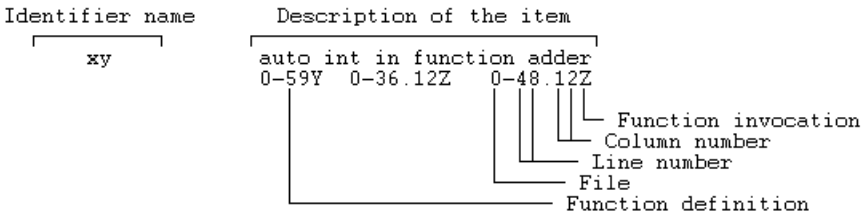
-qnoxref

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 28. 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.
{ and }	Coordinates of the { and } symbols in a structure definition.

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:

```
bgxlc myprogram.c -qxref=full
```

Related information

- "-qattr" on page 74
- "#pragma mc_func" on page 331

-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, -ydn

Parameters

The following suboptions are valid for binary floating-point types only:

m Round toward minus infinity.

n Round to the nearest representable number, ties to even.

p Round toward plus infinity.

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).

Predefined macros

None.

Examples

To compile `myprogram.c` so that constant floating-point expressions are rounded toward zero at compile time, enter:

```
bgxlc myprogram.c -yz
```

Chapter 4. Compiler pragmas reference

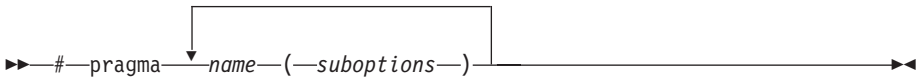
The following sections describe the pragmas available:

Pragma directive syntax

XL C/C++ supports three forms of pragma directives:

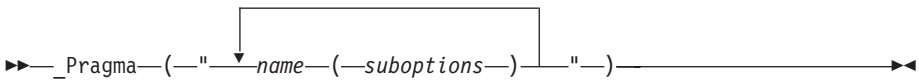
#pragma options *option_name*
: These pragmas use exactly the same syntax as their command-line option equivalent. The exact syntax and list of supported pragmas of this type are provided in "#pragma options" on page 334.

#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:

For descriptions of these categories, see “Summary of compiler options by functional category” on page 43.

Language element control

Table 29. 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 331	Allows you to embed a short sequence of machine instructions "inline" within your program source code.
"#pragma options" on page 334	Specifies compiler options in your source program.

C++ template pragmas

Table 30. C++ template pragmas

Pragma	Description
"#pragma define, #pragma instantiate (C++ only)" on page 315	Provides an alternative method for explicitly instantiating a template class.
"#pragma do_not_instantiate (C++ only)" on page 317	Prevents the specified template declaration from being instantiated.
"#pragma implementation (C++ only)" on page 326	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 31. 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 32. Error checking and debugging pragmas

Pragma	Description
"#pragma ibm snapshot" on page 325	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 33. Listings, messages and compiler information pragmas

Pragma	Description
"#pragma report (C++ only)" on page 345	Controls the generation of diagnostic messages.

Optimization and tuning

Table 34. Optimization and tuning pragmas

Pragma	Description
"#pragma block_loop" on page 310	Marks a block with a scope-unique identifier.
"#pragma STDC cx_limited_range" on page 348	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 315	Lists identifiers that are not aliased to each other within the scope of their use.
"#pragma execution_frequency" on page 318	Marks program source code that you expect will be either very frequently or very infrequently executed.
"#pragma expected_value" on page 320	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 ibm iterations" on page 322	Specifies the approximate average number of loop iterations for the chosen loop.
"#pragma ibm max_iterations" on page 323	Specifies the approximate maximum number of loop iterations for the chosen loop.
"#pragma ibm min_iterations" on page 324	Specifies the approximate minimum number of loop iterations for the chosen loop.
#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 327	Informs the compiler that a named function never returns to the instruction following a call to that function.
"#pragma loopid" on page 328	Marks a block with a scope-unique identifier.
#pragma nosimd	When used with **-qsimd=auto**, disables the generation of SIMD instructions for the next loop.
#pragma novector	When used with **-qhot=vector**, disables auto-vectorization of the next loop.
"#pragma option_override" on page 336	Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.

Table 34. Optimization and tuning pragmas (continued)

Pragma	Description
"#pragma reachable" on page 343	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 344	Specifies registers that may be altered by functions specified by **#pragma mc_func**.
"#pragma simd_level" on page 347	Controls the compiler code generation of vector instructions for individual loops.
"#pragma speculative for" on page 375	Instructs the compiler to speculatively parallelize a for loop.
"#pragma speculative section, #pragma speculative sections" on page 377	The **speculative sections** directive instructs the compiler to speculatively parallelize sections of the code. In code blocks delimited by **speculative sections**, you can use the **speculative section** directive to delimit program code segments.
"#pragma stream_unroll" on page 349	When optimization is enabled, breaks a stream contained in a `for` loop into multiple streams.
"#pragma tm_atomic" on page 379	Indicates a transactional atomic region.
#pragma unroll	Controls loop unrolling, for improved performance.
"#pragma unrollandfuse" on page 350	Instructs the compiler to attempt an unroll and fuse operation on nested `for` loops.

Object code control

Table 35. 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 313	Places a comment into the object module.
"#pragma hashome (C++ only)" on page 321	Informs the compiler that the specified class has a home module that will be specified by **#pragma ishome**.
"#pragma ishome (C++ only)" on page 326	Informs the compiler that the specified class's home module is the current compilation unit.
"#pragma map" on page 329	Converts all references to an identifier to another, externally defined identifier.
"#pragma pack" on page 338	Sets the alignment of all aggregate members to a specified byte boundary.

Table 35. Object code control pragmas (continued)

Pragma	Description
#pragma priority (C++ only)	Specifies the priority level for the initialization of static objects.
"#pragma reg_killed_by" on page 344	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 352	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 36. Portability and migration pragmas

Pragma	Description
#pragma align	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.

Deprecated directives

The SMP directives listed in the following table have been deprecated and might be removed in the future release. Use the corresponding OpenMP directives to obtain the same behavior.

Table 37. Deprecated SMP directives

SMP directive name	OpenMP directive/clause name
#pragma ibm critical	"#pragma omp critical" on page 370
#pragma ibm parallel_loop	The "#pragma omp parallel for" on page 367 pragma with the **schedule** clause.
#pragma ibm schedule	

The following examples show how to replace the deprecated SMP directives with their corresponding OpenMP ones.

For the **critical** pragma:

```
#pragma ibm critical(lck)
{
  ...
}
```

is replaced by

```
#pragma omp critical(lck)
{
...
}
```

For the **schedule** pragma:

```
#pragma ibm parallel_loop
#pragma ibm schedule(static, 5)
for (i=0; i<N; i++)
{
  ...
}
```

is replaced by

```
#pragma omp parallel for schedule(static, 5)
for (i=0; i<N; i++)
{
  ...
}
```

Compiler customization

Table 38. Compiler customization pragmas

Pragma	Description
#pragma complexgcc	Specifies whether to use GCC parameter-passing conventions for complex data types (equivalent to enabling **-qfloat=complexgcc**).

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 303 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 66.

#pragma alloca (C only)

See "-qalloca, -ma (C only)" on page 68.

#pragma block_loop

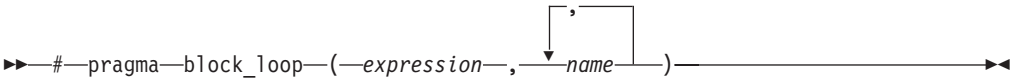
Category

Optimization and tuning

Purpose

Marks a block with a scope-unique identifier.

Syntax



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 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++)
      {
         ...
      }
    }
  }
```

```
#pragma block_loop(50, mymainloop)
#pragma block_loop(20, myfirstloop, mysecondloop)
#pragma loopid(mymainloop)
       for (i=0; i < n; n++)
       {
#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 328
- "-qunroll" on page 286
- "#pragma unrollandfuse" on page 350
- "#pragma stream_unroll" on page 349

#pragma chars

See "-qchars" on page 81.

#pragma comment

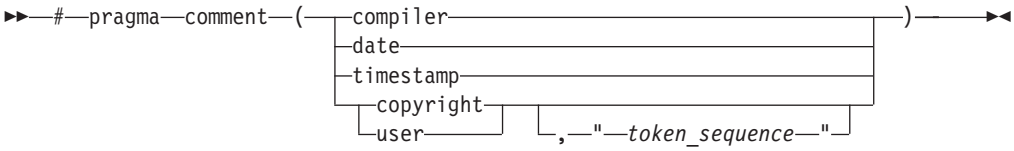
Category

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 we have the following program code: `tt.c`:

```
#pragma comment(date)
#pragma comment(compiler)
#pragma comment(timestamp)
#pragma comment(copyright,"My copyright")
int main() { return 0; }
```

Issuing the command:

```
bgxlc -c tt.c
strings -a tt.o
```

will cause the comment information embedded in `tt.o` to be displayed, along with any other strings that may be found in `tt.o`. For example, assuming the program code shown above:

```
.shstrtab
.strtab
.text
.data
.bss
.tdata
.rela.tdata
.tbss
.rela.text
.rela.data
.symtab
.rodata
.rela.rodata
.toc
.rela.toc
.opd
.rela.opd
.except
.rela.except
.comment
.eh_frame
.rela.eh_frame
Mon Oct 31 10:32:44 2011
IBM XL C/C++ for Blue Gene, Version 12.1.0.0
Mon Oct 31 10:32:31 2011IBM XL C/C++ for Blue Gene, Version 12.1.0.0Mon Oct 31 1     0:32:44 2011
main
My copyright
.The_Code
main
.toc
_$STATIC
```

#pragma complexgcc

See “-qcomplexgccincl” on page 87.

#pragma define, #pragma instantiate (C++ only)

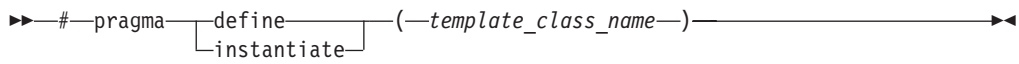
Category

Template control

Purpose

Provides an alternative method for explicitly instantiating a template class.

Syntax



Parameters

template_class_name
The name of the template class to be instantiated.

Usage

This pragma provides the equivalent functionality to C++ explicit instantiation definitions. It is provided for compatibility with earlier releases only. New applications should use C++ explicit instantiation definitions.

This pragma can appear anywhere an explicit instantiation definition 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 317

#pragma disjoint

Category

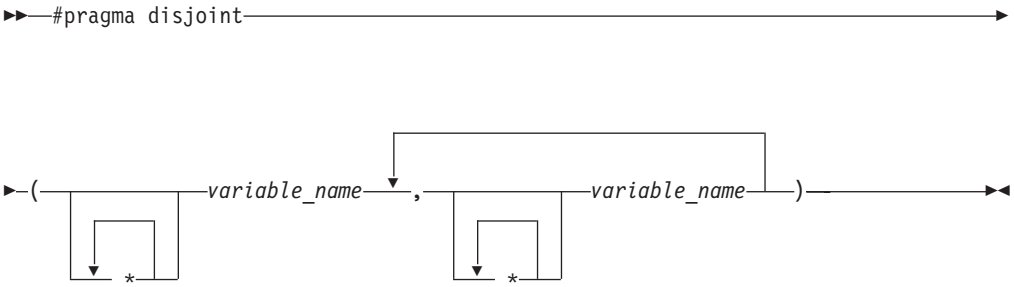
Optimization and tuning

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

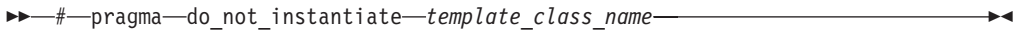
Template control

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



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.

C++0x

Note: C++0x is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++0x standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++0x standard and therefore they should not be relied on as a stable programming interface.

#pragma do_not_instantiate on a class template specialization is treated as an explicit instantiation declaration of the template. This pragma provides a subset of the functionality of the explicit instantiation declarations feature, which is introduced by the C++0x standard. It is provided for compatibility purposes only and is not recommended. New applications should use explicit instantiation declarations instead.

C++0x

You can also use the **-qtmplinst** option to suppress implicit instantiation of template declarations for multiple compilation units. See "-qtmplinst (C++ only)" on page 282.

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 315
- "-qtmplinst (C++ only)" on page 282
- "Explicit instantiation" in the *XL C/C++ Language Reference*
- "-qtempinc (C++ only)" on page 273
- "-qtemplateregistry (C++ only)" on page 276

#pragma enum

See "-qenum" on page 102.

#pragma execution_frequency

Category

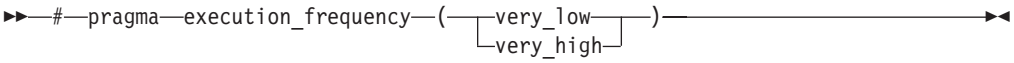
Optimization and tuning

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



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

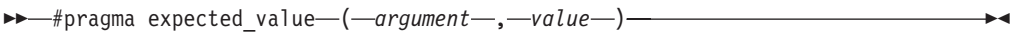
Category

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



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 318

#pragma hashome (C++ only)

Category

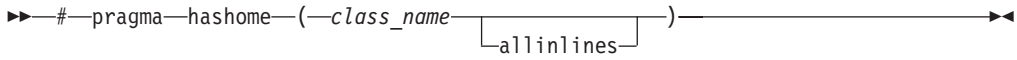
Object code control

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



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 326

#pragma ibm iterations

Category

Optimization and tuning

Purpose

The **iterations** pragma specifies the approximate average number of loop iterations for the chosen loop.

Syntax

Parameters

iteration_count
Specifies the approximate number of loop iterations using a positive integral constant expression.

Usage

The compiler uses the information in *iteration_count* for loop optimization. You can specify multiple #pragma ibm iterations(*iteration_count*).

iteration_count specified in #pragma ibm iterations cannot be smaller than *iteration_count* specified in #pragma ibm min_iterations. In addition, it cannot be bigger than *iteration_count* specified in #pragma ibm max_iterations. Otherwise, the inconsistent value is ignored with a message.

Example

```
#pragma ibm     iterations(100)        // Accepted
#pragma ibm min_iterations(150)        // Ignored (150 > 100)
#pragma ibm min_iterations( 30)        // Accepted( 30 < 100)
```

```
#pragma ibm max_iterations( 60)        // Ignored ( 60 < 100)
#pragma ibm     iterations( 20)        // Ignored ( 20 < 30)
#pragma ibm max_iterations(500)        // Accepted(500 > 100 > 30)
#pragma ibm max_iterations(620)        // Ignored (Multiple occurrences)
#pragma ibm     iterations(200)        // Accepted( 30 < 200 < 500)
#pragma ibm min_iterations( 15)        // Ignored (Multiple occurrences)

   for (int i=0; i < n; ++i)
   {
     #pragma ibm max_iterations(130)   // Accepted
     #pragma ibm min_iterations( 90)   // Accepted( 90 < 130)
     #pragma ibm     iterations( 60)   // Ignored ( 60 <  90)
     #pragma ibm     iterations(100)   // Accepted( 90 < 100 < 130)

     for (int j=0; j < m; ++j) b[j] += a[i];
   }
```

Related reference:

"#pragma ibm max_iterations"

"#pragma ibm min_iterations" on page 324

#pragma ibm max_iterations

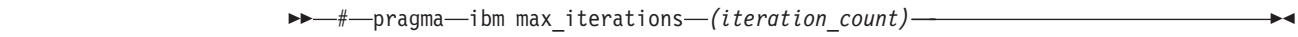
Category

Optimization and tuning

Purpose

The **max_iterations** pragma specifies the approximate maximum number of loop iterations for the chosen loop.

Syntax



Parameters

iteration_count
Specifies the approximate number of maximum loop iterations using a positive integral constant expression.

Usage

The compiler uses the information in *iteration_count* for loop optimization. You can specify #pragma ibm max_iterations(*iteration_count*) only once. If you specify #pragma ibm max_iterations(*iteration_count*) more than once, the first specified pragma is accepted, and the subsequent pragmas are ignored with a message.

iteration_count specified in #pragma ibm max_iterations cannot be smaller than *iteration_count* specified in #pragma ibm iterations or #pragma ibm min_iterations. Otherwise, the inconsistent value is ignored with a message.

Example

```
#pragma ibm     iterations(100)        // Accepted
#pragma ibm min_iterations(150)        // Ignored (150 > 100)
#pragma ibm min_iterations( 30)        // Accepted( 30 < 100)
#pragma ibm max_iterations( 60)        // Ignored ( 60 < 100)
#pragma ibm     iterations( 20)        // Ignored ( 20 < 30)
```

```
#pragma ibm max_iterations(500)        // Accepted(500 > 100 > 30)
#pragma ibm max_iterations(620)        // Ignored (Multiple occurrences)
#pragma ibm     iterations(200)        // Accepted( 30 < 200 < 500)
#pragma ibm min_iterations( 15)        // Ignored (Multiple occurrences)

   for (int i=0; i < n; ++i)
   {
     #pragma ibm max_iterations(130)   // Accepted
     #pragma ibm min_iterations( 90)   // Accepted( 90 < 130)
     #pragma ibm     iterations( 60)   // Ignored ( 60 <  90)
     #pragma ibm     iterations(100)   // Accepted( 90 < 100 < 130)

     for (int j=0; j < m; ++j) b[j] += a[i];
   }
```

Related reference:

"#pragma ibm iterations" on page 322

"#pragma ibm min_iterations"

#pragma ibm min_iterations

Category

Optimization and tuning

Purpose

The **min_iterations** pragma specifies the approximate minimum number of loop iterations for the chosen loop.

Syntax

Parameters

iteration_count
Specifies the approximate minimum number of loop iterations using a positive integral constant expression.

Usage

The compiler uses the information in *iteration_count* for loop optimization. You can specify #pragma ibm min_iterations(*iteration_count*) only once. If you specify #pragma ibm min_iterations(*iteration_count*) more than once, the first specified pragma is accepted, and the subsequent pragmas are ignored with a message.

iteration_count specified in #pragma ibm min_iterations cannot be bigger than *iteration_count* specified in #pragma ibm iterations or #pragma ibm max_iterations. Otherwise, the inconsistent value is ignored with a message.

Example

```
#pragma ibm     iterations(100)        // Accepted
#pragma ibm min_iterations(150)        // Ignored (150 > 100)
#pragma ibm min_iterations( 30)        // Accepted( 30 < 100)
#pragma ibm max_iterations( 60)        // Ignored ( 60 < 100)
#pragma ibm     iterations( 20)        // Ignored ( 20 < 30)
#pragma ibm max_iterations(500)        // Accepted(500 > 100 > 30)
#pragma ibm max_iterations(620)        // Ignored (Multiple occurrences)
```

```
#pragma ibm     iterations(200)        // Accepted( 30 < 200 < 500)
#pragma ibm min_iterations( 15)        // Ignored (Multiple occurrences)

   for (int i=0; i < n; ++i)
   {
     #pragma ibm max_iterations(130)   // Accepted
     #pragma ibm min_iterations( 90)   // Accepted( 90 < 130)
     #pragma ibm     iterations( 60)   // Ignored ( 60 <  90)
     #pragma ibm     iterations(100)   // Accepted( 90 < 100 < 130)

     for (int j=0; j < m; ++j) b[j] += a[i];
   }
```

Related reference:

"#pragma ibm iterations" on page 322

"#pragma ibm max_iterations" on page 323

#pragma ibm snapshot

Category

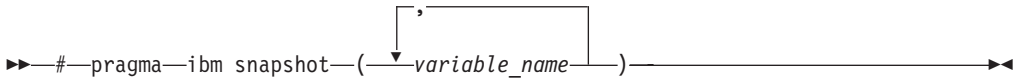
Error checking and debugging

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



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.

#pragma ibm snapshot is not supported in transactional atomic regions.

Examples

```
#pragma ibm snapshot(a, b, c)
```

Related information

- "-g" on page 121
- "-O, -qoptimize" on page 207

#pragma implementation (C++ only)

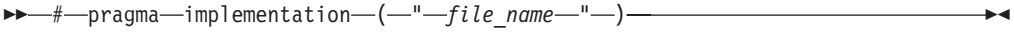
Category

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



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 273
- "Using C++ templates"

#pragma info

See "-qinfo" on page 139.

#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

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 321

Related information

- "#pragma hashome (C++ only)" on page 321

#pragma isolated_call

See "-qisolated_call" on page 157.

#pragma langlvl (C only)

See "-qlanglvl" on page 165.

#pragma leaves

Category

Optimization and tuning

Purpose

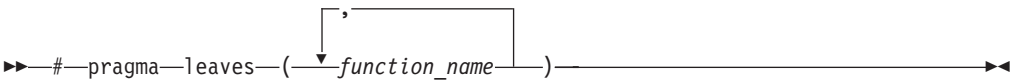
Informs the compiler that a named function never returns to the instruction following a call to that function.

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



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 343.

#pragma loopid

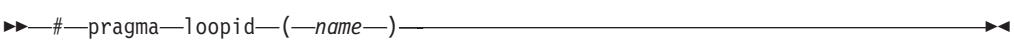
Category

Optimization and tuning

Purpose

Marks a block with a scope-unique identifier.

Syntax



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 310.

Related information

- "-qunroll" on page 286
- "#pragma block_loop" on page 310
- "#pragma unrollandfuse" on page 350

#pragma map

Category

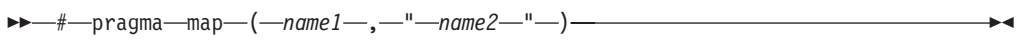
Object code control

Purpose

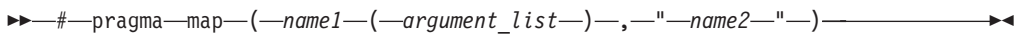
Converts all references to an identifier to another, externally defined identifier.

Syntax

#pragma map syntax – C



#pragma map syntax – C++



Parameters

name1

The name used in the source code. C *name1* can represent a data object or function with external linkage. C++ *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!
```

C++ 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

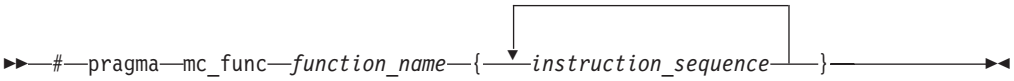
Language element control

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



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 344 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**.

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 157
- "#pragma reg_killed_by" on page 344
- "Inline assembly statements" in the *XL C/C++ Language Reference*

#pragma nofunctrace

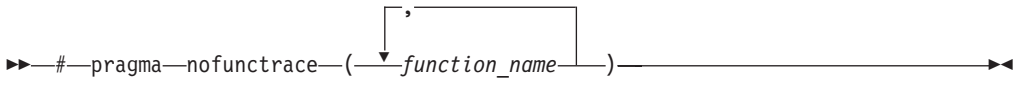
Category

Error checking and debugging

Purpose

Disables tracing for a given function or a list of specified functions.

Syntax



Parameters

function_name
The name of the function for which you want to disable tracing.

Usage

When you use **#pragma nofunctrace** to specify a list of functions for which you want to disable tracing, use parenthesis **()** and encapsulate the functions in it. For a list of functions, use a comma **,** to separate them. For example, to disable tracing for function `a`, use `#pragma nofunctrace(a)`. To disable tracing for functions `a`, `b`, and `c`, use `#pragma nofunctrace(a,b,c)`.

If you have two functions: `foo(int)` and `foo(double)`, use `#pragma nofunctrace(foo(int))` disables tracing for `foo(int)` but not `foo(double)`.

Two colons in a row `::` are considered scope qualifiers. For example, when you call `-qfunctrace+A::B:C`, the compiler traces functions that begin with the qualifiers `A::B` or `C`.

Note: If you want to use the compiler option **-qfunctrace** to disable tracing for a given function or a list of functions, you must use its suboption **-** followed by the

names of the functions. For details about how to use **-qfunctrace** and its related suboptions, see "-qfunctrace" on page 118.

Examples

```
#pragma nofunctrace(a,b,c)
```

Related information

- "-qfunctrace" on page 118

#pragma nosimd

See "-qhot" on page 130.

#pragma novector

See "-qhot" on page 130.

#pragma options

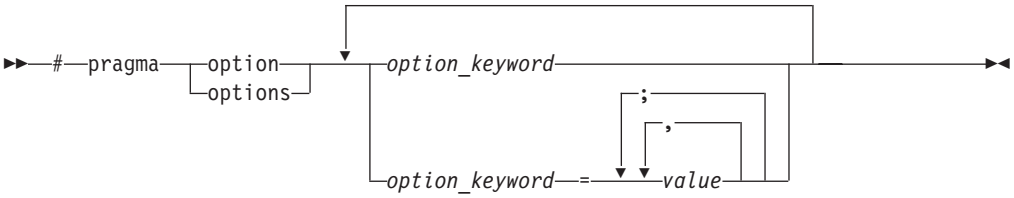
Category

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, see the pages of the equivalent compiler option.

Valid settings for #pragma options ***option_keyword***	**Compiler option equivalent**
align=*option*	"-qalign" on page 66
[no]attr attr=full	"-qattr" on page 74
chars=*option*	"-qchars" on page 81
[no]check	"-qcheck" on page 83
[no]compact	"-qcompact" on page 86
[no]dbcs	"-qmbcs, -qdbcs" on page 202
[no]digraph	"-qdigraph" on page 96
[no]dollar	"-qdollar" on page 97

Valid settings for #pragma options *option_keyword*	**Compiler option equivalent**
enum=*option*	"-qenum" on page 102
flag=*option*	"-qflag" on page 107
float=[no]*option*	"-qfloat" on page 109
[no]flttrap	"-qflttrap" on page 113
[no]fullpath	"-qfullpath" on page 118
halt	"-qhalt" on page 127
[no]idirfirst	"-qidirfirst" on page 134
[no]ignerrno	"-qignerrno" on page 135
ignprag=*option*	"-qignprag" on page 136
[no]info=*option*	"-qinfo" on page 139
initauto=*value*	"-qinitauto" on page 146
isolated_call=*names*	"-qisolated_call" on page 157
C langlvl	"-qlanglvl" on page 165
[no]ldbl128	"-qldbl128" on page 186
[no]libansi	"-qlibansi" on page 188
[no]list	"-qlist" on page 191
[no]longlong	"-qlonglong" on page 197
[no]maxmem=*number*	"-qmaxmem" on page 201
[no]mbcs	"-qmbcs, -qdbcs" on page 202
[no]optimize=*number*	"-O, -qoptimize" on page 207
C++ priority=*number*	"-qpriority (C++ only)" on page 224
proclocal, procimported, procunknown	"-qprocimported, -qproclocal, -qprocunknown" on page 225
C [no]proto	"-qproto (C only)" on page 227
[no]ro	"-qro" on page 234
[no]roconst	"-qroconst" on page 235
[no]showinc	"-qshowinc" on page 241
[no]source	"-qsource" on page 251
spill=*number*	"-qspill" on page 254
[no]stdinc	"-qstdinc" on page 260
[no]strict	"-qstrict" on page 261
tbtable=*option*	"-qtbtable" on page 272
tune=*option*	"-qtune" on page 284
[no]unrollunroll=*number*	"-qunroll" on page 286
C [no]upconv	"-qupconv (C only)" on page 290
[no]xref	"-qxref" on page 299

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

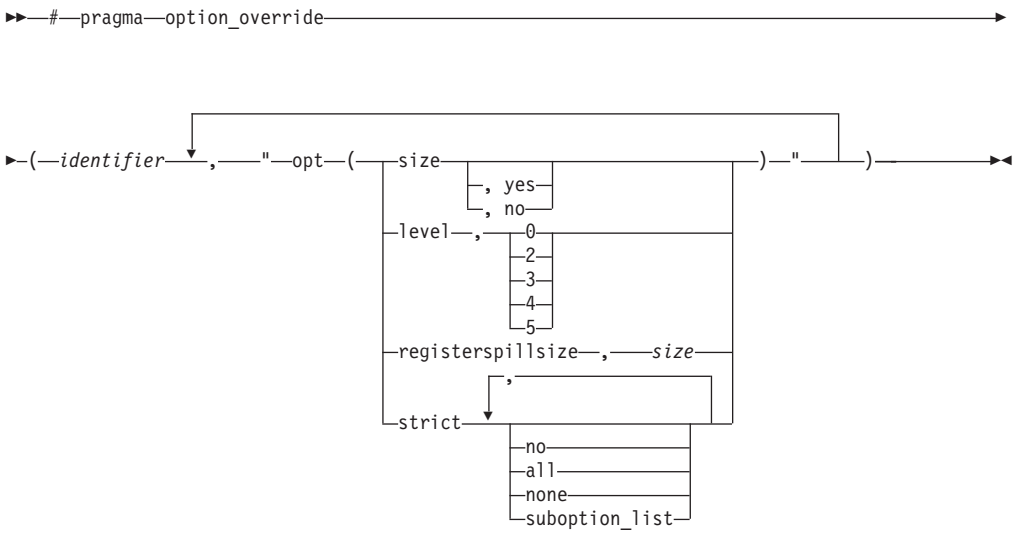
Optimization and tuning

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, *size*	-qspill=*size*
size	-qcompact
size, yes	
size, no	-qnocompact
strict, all	-qstrict, -qstrict=all
strict, no, none	-qnostrict
strict, *suboption_list*	-qstrict=*suboption_list*

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.

C++ 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 207
- "-qcompact" on page 86
- "-qspill" on page 254
- "-qstrict" on page 261

#pragma pack

Category

Object code control

Purpose

Sets the alignment of all aggregate members to a specified byte boundary.

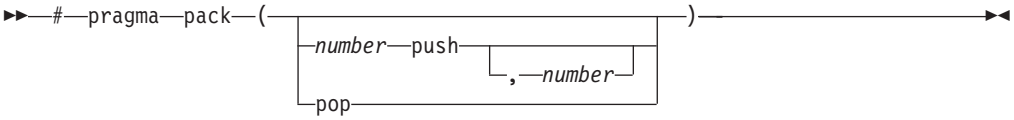
If the byte boundary number is smaller than the natural alignment of a member, padding bytes are removed, thereby reducing the overall structure or union size.

The syntax and semantics of this pragma are different depending on the setting of the **-qpack_semantic** option.

Syntax

Default #pragma pack syntax (-qpack_semantic=ibm in effect)

#pragma pack syntax with -qpack_semantic=gnu in effect



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.

push
When specified without a *number*, pushes whatever value is currently in effect to the top of the packing "stack". When used with a *number*, pushes that value to the top of the packing stack, and sets the packing value to that of *number* for

structures that follow. Note that this parameter is not recognized when **-qpack_semantic=ibm** 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
When **-qpack_semantic=ibm** is in effect, sets the packing rule to that which was in effect before the current setting. When **-qpack_semantic=gnu** is in effect, pops the value specified in the last **push** statement off the stack and resets the current packing value to the value on the top of the stack, overriding any intervening value that may have been specified without a **push** statement.

Specifying **#pragma pack()** with no parameters (that is, with empty parentheses) has the following effect:

- Disables all packing (equivalent to specifying **#pragma pack(nopack)**), when **-qpack_semantic=ibm** is in effect.
- Sets the current packing value to that which was in effect at the beginning of the compilation unit, when **-qpack_semantic=gnu** is in effect.

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 gives members of aggregates an alignment of 1 byte. When **#pragma pack** is applied to an aggregate with a `vector4double` member, the compiler generates a severe error message.

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:

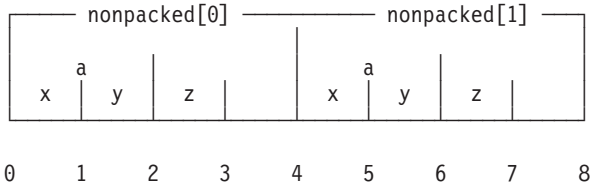
```
struct s_t {
 char a;
 int b;
 short c;
 int d;
}S;
```

Default mapping:	**With #pragma pack(1):**
size of s_t = 16	size of s_t = 11
offset of a = 0	offset of a = 0
offset of b = 4	offset of b = 1
offset of c = 8	offset of c = 5
offset of d = 12	offset of d = 7
alignment of a = 1	alignment of a = 1
alignment of b = 4	alignment of b = 1
alignment of c = 2	alignment of c = 1
alignment of d = 4	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:

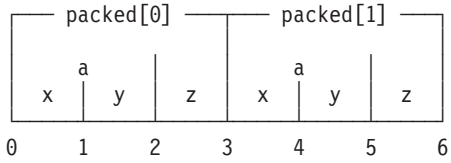


The next example uses **#pragma pack(1)** to set the alignment of unions of type uu to 1 byte:

```
#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:



The following examples show the results of the differences in the semantics of this pragma depending on whether **-qpack_semantic=ibm** or **-qpack_semantic=ibm** is in effect.

This example shows the effect of specifying the **push** parameter:

```
#pragma pack(1)
#pragma pack(push)    // ignored when -qpack_semantic=ibm is in effect
#pragma pack(push,2)  // ignored when -qpack_semantic=ibm is in effect

struct s_t {
   char a;
   int b;
} S;
```

With **-qpack_semantic=gnu** in effect, the packing value in effect when the structure `S` is declared is 2, and the structure is aligned on 2-byte boundaries. With **-qpack_semantic=ibm** in effect, the second two directives are ignored, and the packing value in effect for structure `S` is 1, and it is aligned on 1-byte boundaries.

This example shows the effect of specifying the **push** and **pop** parameters together:

```
#pragma pack(push,1)    // ignored when -qpack_semantic=ibm is in effect
#pragma pack(push,4)    // ignored when -qpack_semantic=ibm is in effect
#pragma pack(2)
#pragma pack(pop)
#pragma pack(pop)
#pragma pack(pop)

struct s_t {
   char a;
   int b;
} S;
```

With **-qpack_semantic=gnu** in effect, since **pop** only pops values that have been pushed onto the stack with a **push** directive, the first **pop** directive pops 4 off the stack, the second one pops 1 off the stack, and the alignment is the setting in effect at the beginning of the compilation unit (the intervening **#pragma pack(2)** directive is overridden). With **-qpack_semantic=ibm** in effect, the **pop** statement pops the value 2 off the stack, and the alignment is the setting in effect at the beginning of the compilation unit.

The following example shows the effect of specifying the directive inside a nested structure:

```
struct s_t {
   char a;
   int b;

   #pragma pack(1)

   struct t_t {
      char x;
      int y;
   }T;

   char c;

   #pragma pack(2)
   #pragma pack(1)

   int d;

   #pragma align(natural)   \\ this only affects u_t.
   #pragma pack(2)          \\ this only affects u_t.

   struct u_t {
      char j;
      int k;
   }U;

}S;
```

When **-qpack_semantic=gnu** is in effect, the first **#pragma pack(1)** directive applies to both structure `t_t` *and* `s_t`. With **-qpack_semantic=ibm** the first **#pragma pack(1)** directive applies to structure `t_t` only.

Related information

- "-qalign" on page 66
- "-qpack_semantic" on page 216
- "Using alignment modifiers" in the *XL C/C++ Optimization and Programming Guide*

#pragma priority (C++ only)

See "-qpriority (C++ only)" on page 224.

#pragma reachable

Category

Optimization and tuning

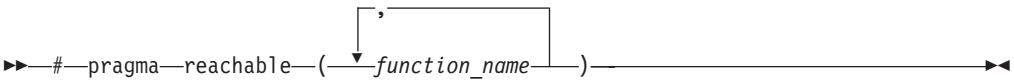
Purpose

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



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 327

#pragma reg_killed_by

Category

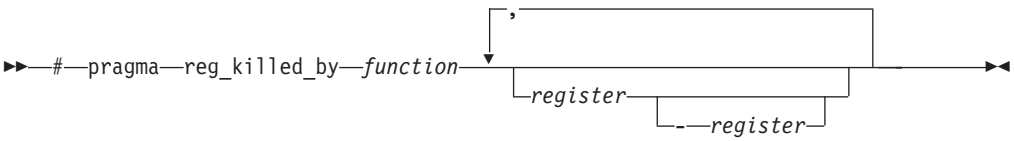
Optimization and tuning

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 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 (on selected processors only)

xer Fixed-point exception register

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 331

#pragma report (C++ only)

Category

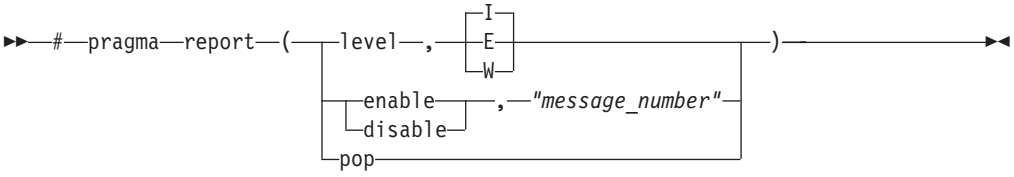
Listings, messages and compiler information

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 in quotation marks; for example, "CCN1004".

Note: You must use quotation marks with *message_number* as in the preceding 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 107

#pragma simd_level

Category

Optimization and tuning

Purpose

Controls the compiler code generation of vector instructions for individual loops.

Vector instructions can offer high performance when used with algorithmic-intensive tasks such as multimedia applications. You have the flexibility to control the aggressiveness of autosimdization on a loop-by-loop basis, and might be able to achieve further performance gain with this fine grain control.

The supported levels are from 0 to 10. level(0) indicates performing no autosimdization on the loop that follows the pragma directive. level(10) indicates performing the most aggressive form of autosimdization on the loop. With this pragma directive, you can control the autosimdization behavior on a loop-by-loop basis.

Syntax

Parameters

n A scalar integer initialization expression, from 0 to 10, specifying the aggressiveness of autosimdization on the loop that follows the pragma directive.

Usage

A loop with no simd_level pragma is set to simd level 5 by default, if **-qsimd=auto** is in effect.

#pragma simd_level(0) is equivalent to **#pragma nosimd**, where autosimdization is not performed on the loop that follows the pragma directive.

#pragma simd_level(10) instructs the compiler to perform autosimdization on the loop that follows the pragma directive most aggressively, including bypassing cost analysis.

Rules

The rules of **#pragma simd_level** directive are listed as follows:

- The **#pragma simd_level** directive has effect only for architectures that support vector instructions and when used with **-qsimd=auto**.
- The **#pragma simd_level** directive applies only to the loop immediately following it. The directive has no effect on other loops that are nested within the specified loop. It is possible to set different simd levels for the inner and outer loops by specifying separate **#pragma simd_level** directives.
- The **#pragma simd_level** directive can be mixed with loop optimization (**-qhot**) and OpenMP directives without requiring any specific optimization level. For

more information about **-qhot** and OpenMP directives, see "-qhot" on page 130 in this document and "Using OpenMP directives" in the *IBM XL C/C++ Optimization and Programming Guide*.

Examples

```
...
#pragma simd_level(10)
for (i=1; i<1000; i++) {
/* program code */

} ...
```

Related information

•

#pragma STDC cx_limited_range

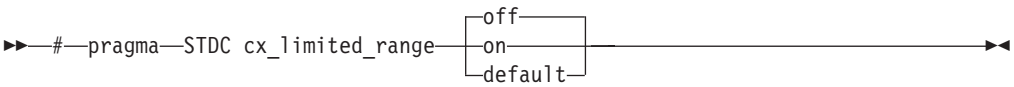
Category

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



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

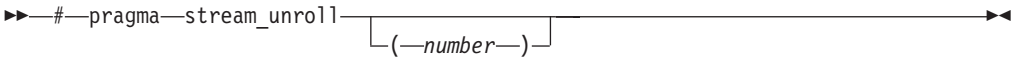
Category

Optimization and tuning

Purpose

When optimization is enabled, breaks a stream contained in a `for` loop into multiple streams.

Syntax



Parameters

number

A loop unrolling factor. C The value of *number* is a positive integral constant expression. C++ 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 **-qsmp**, 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. C 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. C++ The compiler silently ignores all but the last of multiple loop unrolling pragmas specified on the same `for` loop.

Examples

The following example shows how **#pragma stream_unroll** can increase performance.

```
int i, m, n;
int a[1000];
int b[1000];
int c[1000];

....

#pragma stream_unroll(4)
for (i=0; 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:

```
m = n/4;

for (i=0; 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];
}
```

The increased number of read and store operations are distributed among a number of streams determined by the compiler, which reduces computation time and increase performance.

Related information

- "-qunroll" on page 286
- "#pragma unrollandfuse"

#pragma strings

See "-qro" on page 234.

#pragma unroll

See "-qunroll" on page 286.

#pragma unrollandfuse

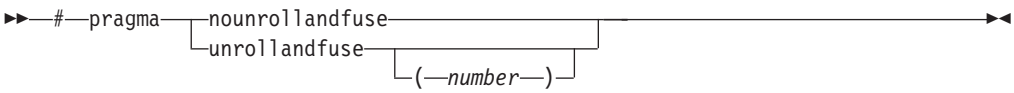
Category

Optimization and tuning

Purpose

Instructs the compiler to attempt an unroll and fuse operation on nested `for` loops.

Syntax



Parameters

number

A loop unrolling factor. C The value of *number* is a positive integral constant expression. C++ The value of *number* is a positive scalar integer or compile-time constant initialization expression.

If *number* is not specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Usage

The **#pragma unrollandfuse** directive applies only to the outer loops of nested `for` loops that 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.

For loop unrolling to occur, the **#pragma unrollandfuse** directive must precede a `for` loop. You must not specify **#pragma unrollandfuse** for the innermost `for` loop.

You must not specify **#pragma unrollandfuse** more than once, or combine the directive with **#pragma nounrollandfuse**, **#pragma nounroll**, **#pragma unroll**, or **#pragma stream_unroll** directives for the same `for` loop.

Predefined macros

None.

Examples

In the following example, a **#pragma unrollandfuse** directive replicates and fuses the body of the loop. This reduces the number of cache misses for array `b`.

```
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 286
- "#pragma stream_unroll" on page 349

#pragma weak

Category

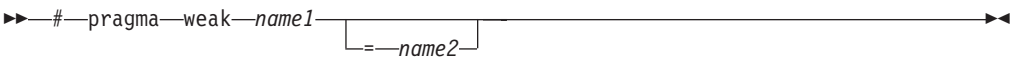
Object code control

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



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 _Z3foov
#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

#if __cplusplus
#pragma weak _Z3foov = _Z4foo2v
#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 329

Pragma directives for parallel processing

Parallel processing operations are controlled by pragma directives in your program source.

- The `omp` pragmas have effect only when parallelization is enabled with the **-qsmp** compiler option.
- The `speculative` pragmas have effect only when thread-level speculative execution is enabled with the **-qsmp=speculative** compiler option.
- The `tm` pragmas have effect only when transactional memory is enabled with the **-qtm** compiler option.

Nesting OpenMP, transactional memory, and thread-level speculative execution

This section describes how you can mix parallel regions. The following types of parallel regions can be used without restrictions in the same program if they are not nested:

- OpenMP
- Transactional memory (TM)
- Thread-level speculative execution (SE)

They can also be nested but with some restrictions. The following table describes the behavior of different nesting scenarios.

Table 39. Nesting rules for OpenMP, TM, and SE

Scenario	Description	Runtime action

`BEGIN SE`
`  BEGIN SE`
`  END SE`
`END SE`
	An SE region is nested inside an SE region.	The SE nesting is flattened. The inner nested SE region is run speculatively by one thread as part of the parallel outer SE region. The clauses specified on the inner SE region are still effective.

`BEGIN SE`
`  BEGIN TM`
`  END TM`
`END SE`
	A TM region is nested inside an SE region.	The TM region is run speculatively in SE mode as part of the outer SE region.

`BEGIN SE`
`  BEGIN OpenMP`
`  END OpenMP`
`END SE`
	An OpenMP region is nested inside an SE region.	An OpenMP region running in parallel inside the speculative SE region causes the SE region to be stopped. The stopped SE region is rolled back and run nonspeculatively. The inner OpenMP region is run nonspeculatively by multiple threads.

`BEGIN TM`
`  BEGIN SE`
`  END SE`
`END TM`
	An SE region is nested inside a TM region.	The inner SE region is run speculatively in TM mode by one thread as part of the outer TM region.

`BEGIN TM`
`  BEGIN TM`
`  END TM`
`END TM`
	A TM region is nested inside a TM region.	The TM nesting is flattened. The inner nested TM regions are run speculatively as part of the outer TM region.

`BEGIN TM`
`  BEGIN OpenMP`
`  END OpenMP`
`END TM`
	An OpenMP region is nested inside a TM region.	An OpenMP region running in parallel inside the speculative TM region causes the TM region to be stopped. The stopped transaction is then rolled back and run nonspeculatively. The inner OpenMP region is run nonspeculatively by multiple threads.

`BEGIN OpenMP`
`  BEGIN SE`
`  END SE`
`END OpenMP`
	An SE region is nested inside an OpenMP region.	The SE region is run by one thread in nonspeculative mode. The outer OpenMP region is run in parallel.

`BEGIN OpenMP`
`  BEGIN TM`
`  END TM`
`END OpenMP`
	A TM region is nested inside an OpenMP region.	The outer OpenMP region is run in parallel and the TM region is run speculatively.

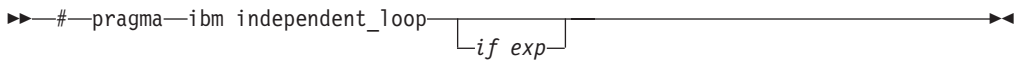
`BEGIN TM`
`END TM`
 	
`BEGIN SE`
`END SE`
	A program contains separate TM and SE regions.	The first region is run in TM mode. The second region is run in SE mode by one thread.
		
		If the `reset_speculation_mode` function is called after the first TM region, the second SE region is run in parallel speculatively.

`BEGIN SE`
`END SE`
 	
`BEGIN TM`
`END TM`
	A program contains separate SE and TM regions.	The first region is run in SE mode. The second region is run in TM irrevocable mode.
		
		If the `reset_speculation_mode` function is called after the first SE region, the second TM region is run in parallel speculatively.

#pragma ibm independent_loop
Purpose

The **independent_loop** pragma explicitly states that the iterations of the chosen loop are independent, and that the iterations can be executed in parallel.

Syntax



where exp represents a scalar expression.

Usage

If the iterations of a loop are independent, you can put the pragma before the loop block. Then the compiler executes these iterations in parallel. When the exp argument is specified, the loop iterations are considered independent only if exp evaluates to TRUE at run time.

Notes:

- If the iterations of the chosen loop are dependent, the compiler executes the loop iterations sequentially no matter whether you specify the **independent_loop** pragma.
- To have an effect on a loop, you must put the **independent_loop** pragma immediately before this loop. Otherwise, the pragma is ignored.
- If several **independent_loop** pragmas are specified before a loop, only the last one takes effect.

This pragma can be combined with the **omp parallel for** pragma to select a specific parallel process scheduling algorithm. For more information, see "#pragma omp parallel for" on page 367.

Examples

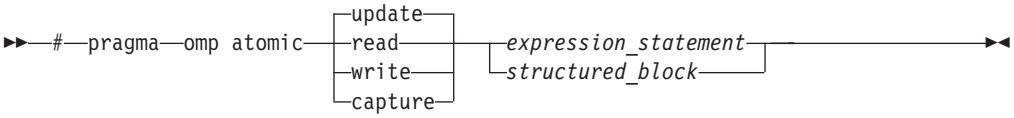
In the following example, the loop iterations are executed in parallel if the value of the argument k is larger than 2.

```
int a[1000], b[1000], c[1000];
int main(int k){
   if(k>0){
      #pragma ibm independent_loop if (k>2)
      for(int i=0; i<900; i++){
         a[i]=b[i]*c[i];
      }
   }
}
```

#pragma omp atomic
Purpose

The **omp atomic** directive allows access of a specific memory location atomically. It ensures that race conditions are avoided through direct control of concurrent threads that might read or write to or from the particular memory location. With the **omp atomic** directive, you can write more efficient concurrent algorithms with fewer locks.

Syntax



where *expression_statement* is an expression statement of scalar type, and *structured_block* is a structured block of two expression statements.

Clauses

update
Updates the value of a variable atomically. Guarantees that only one thread at a time updates the shared variable, avoiding errors from simultaneous writes to the same variable. An **omp atomic** directive without a clause is equivalent to an **omp atomic** update.

Note: Atomic updates cannot write arbitrary data to the memory location, but depend on the previous data at the memory location.

read
Reads the value of a variable atomically. The value of a shared variable can be read safely, avoiding the danger of reading an intermediate value of the variable when it is accessed simultaneously by a concurrent thread.

write
Writes the value of a variable atomically. The value of a shared variable can be written exclusively to avoid errors from simultaneous writes.

capture
Updates the value of a variable while capturing the original or final value of the variable atomically.

The *expression_statement* or *structured_block* takes one of the following forms, depending on the atomic directive clause:

Directive clause	***expression_statement***	***structured_block***

update
(equivalent to no clause)
	`x++;`
	`x--;`
	`++x;`
	`--x;`
	`x binop = expr;`
	`x = x binop expr;`	

read	`v = x;`	
write	`x = expr;`	

Directive clause	*expression_statement*	*structured_block*
capture	`v = x++;`	`{v = x; x binop = expr;}`
	`v = x--;`	`{v = x; xOP;}`
	`v = ++x;`	`{v = x; OPx;}`
	`v = --x;`	`{x binop = expr; v = x;}`
	`v = x binop = expr;`	`{xOP; v = x;}`
		`{OPx; v = x;}`
		`{v = x; x = x binop expr;}`
		`{x = x binop expr; v = x;}`

where:

x, *v* are both lvalue expressions with scalar type.

expr is an expression of scalar type that does not reference *x*.

binop is one of the following binary operators:
`+ * - / & ^ | << >>`

OP is one of `++` or `--`.

Note: *binop*, *binop=*, and *OP* are not overloaded operators.

Usage

Objects that can be updated in parallel and that might be subject to race conditions should be protected with the **omp atomic** directive.

All atomic accesses to the storage locations designated by *x* throughout the program should have a compatible type.

Within an atomic region, multiple syntactic occurrences of *x* must designate the same storage location.

All accesses to a certain storage location throughout a concurrent program must be atomic. A non-atomic access to a memory location might break the expected atomic behavior of all atomic accesses to that storage location.

Neither *v* nor *expr* can access the storage location that is designated by *x*.

Neither *x* nor *expr* can access the storage location that is designated by *v*.

All accesses to the storage location designated by *x* are atomic. Evaluations of the expression *expr*, *v*, *x* are not atomic.

For atomic capture access, the operation of writing the captured value to the storage location represented by *v* is not atomic.

Examples

Example 1: Atomic update

```
extern float x[], *p = x, y;

/* Protect against race conditions among multiple updates. */
#pragma omp atomic
x[index[i]] += y;

/* Protect against race conditions with updates through x. */
#pragma omp atomic
p[i] -= 1.0f;
```

Example 2: Atomic read, write, and update

```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic read
  temp[i] = x[f(i)];

  #pragma omp atomic write
  x[i] = temp[i]*2;

  #pragma omp atomic update
  x[i] *= 2;
}
```

Example 3: Atomic capture

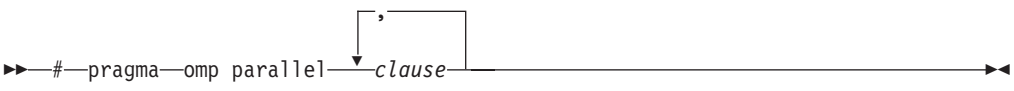
```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic capture
  temp[i] = x[f(i)]++;

  #pragma omp atomic capture
  {
    temp[i] = x[f(i)]; //the two occurences of x[f(i)] must evaluate to the
    x[f(i)] -= 3; //same memory location, otherwise behavior is undefined.
  }
}
```

#pragma omp parallel

Purpose

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen block of code.

Syntax



Parameters

clause is any of the following clauses:

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 explcitly 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 in the original value of the shared reduction variable. For example, when the `max` operator is specified, the original reduction variable value combines with the final values of the private copies by using the following expression:

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

For variables specified in the **reduction** clause, they must satisfy the following conditions:

- Must be of a type appropriate to the operator. If the `max` or `min` operator is specified, the variables must be one of the following types with or without `long`, `short`, `signed`, or `unsigned`:
  - C _Bool C
  - C++ bool C++
  - char
  - C++ wchar_t C++
  - int
  - float
  - double
- 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.

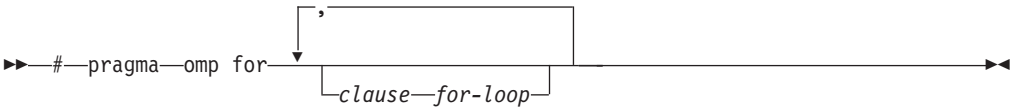
Nested parallel regions are always serialized.

#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 clauses:

collapse (*n*)
Allows you to parallelize multiple loops in a nest without introducing nested parallelism.

- Only one collapse clause is allowed on a worksharing **for** or **parallel for** pragma.
- The specified number of loops must be present lexically. That is, none of the loops can be in a called subroutine.

- The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.
- If the loop indices are of different size, the index with the largest size will be used for the collapsed loop.
- The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP pragma between the loops which are collapsed.
- The associated do-loops must be structured blocks. Their execution must not be terminated by an **break** statement.
- If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop may be curtailed by a **continue** statement. If multiple loops are associated to the loop construct, there must be no branches to any of the loop termination statements except for the innermost associated loop.

Ordered construct
During execution of an iteration of a loop or a loop nest within a loop region, the executing thread must not execute more than one ordered region which binds to the same loop region. As a consequence, if multiple loops are associated to the loop construct by a collapse clause, the ordered construct has to be located inside all associated loops.

Lastprivate clause
When a lastprivate clause appears on the pragma that identifies a work-sharing construct, the value of each new list item from the sequentially last iteration of the associated loops, is assigned to the original list item even if a collapse clause is associated with the loop

Other SMP and performance pragmas
stream_unroll,**unroll**,**unrollandfuse**,**nounrollandfuse** pragmas cannot be used for any of the loops associated with the **collapse** clause loop nest.

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 in the original value of the shared reduction variable. For example, when the `max` operator is specified, the original reduction variable value combines with the final values of the private copies by using the following expression:

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

For variables specified in the **reduction** clause, they must satisfy the following conditions:

- Must be of a type appropriate to the operator. If the `max` or `min` operator is specified, the variables must be one of the following types with or without `long`, `short`, `signed`, or `unsigned`:
  - C _Bool C
  - C++ bool C++
  - char
  - C++ wchar_t C++
  - int
  - float
  - double
- 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 active threads on a "first-come, first-do" basis until all work has been assigned.

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 active threads on a "first-come, first-do" basis until all work has been assigned.

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:

init_expr takes the form:

```
iv = b
integer-type iv = b
```

exit_cond takes the form:

```
iv <= ub
iv <  ub
iv >= ub
iv >  ub
```

incr_expr takes the form:

```
++iv
iv++
--iv
iv--
iv += incr
iv -= incr
iv = iv + incr
iv = incr + iv
iv = iv - incr
```

and where:

iv 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.

b, *ub*, *incr* 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:

- 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** clause.
- 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

Usage

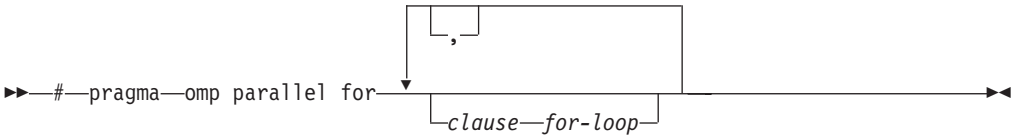
The **omp ordered** directive must be used as follows:

- 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



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
Purpose

The **omp sections** directive distributes work among threads bound to a defined parallel region.

Syntax

Parameters

clause is any of the following clauses:

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 in the original value of the shared reduction variable. For example, when the `max` operator is specified, the original reduction variable value combines with the final values of the private copies by using the following expression:

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

For variables specified in the **reduction** clause, they must satisfy the following conditions:

- Must be of a type appropriate to the operator. If the `max` or `min` operator is specified, the variables must be one of the following types with or without `long`, `short`, `signed`, or `unsigned`:
  - C _Bool C
  - C++ bool C++
  - char
  - C++ wchar_t C++
  - int
  - float
  - double
- 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.

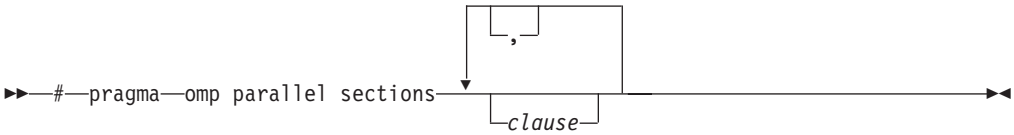
When program execution reaches a **omp sections** directive, program segments defined by the following **omp section** directive are distributed for parallel execution among available threads. A barrier is implicitly defined at the end of the larger program region associated with the **omp sections** 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



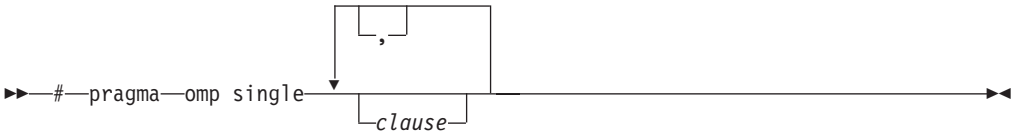
Usage

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

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

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

Usage

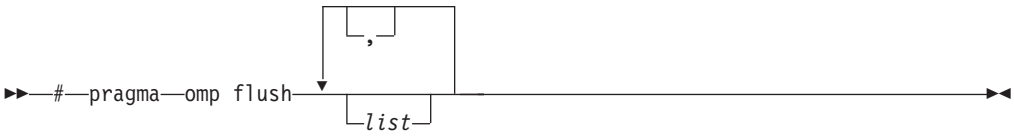
The **omp barrier** directive must appear within a block or compound statement. For example:

```
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:

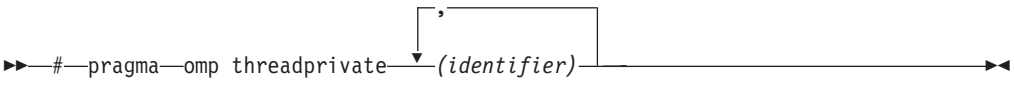
```
if (x!=0) {
   #pragma omp flush    /* valid usage    */
}
if (x!=0)
   #pragma omp flush    /* invalid usage  */
```

#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.

#pragma omp task
Purpose

The **task** pragma can be used to explicitly define a task.

Use the **task** pragma when you want to identify a block of code to be executed in parallel with the code outside the task region. The **task** pragma can be useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms. The **task** directive takes effect only if you specify the **-qsmp** compiler option.

Syntax

Parameters

The *clause* parameter can be any of the following types of clauses:

default (shared | none)
Defines the default data scope of variable in each task. Only one `default` clause can be specified on an `omp task` 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 construct must be explicitly listed in a data scope clause, with the exception of variables with the following attributes:

- Threadprivate
- Automatic and declared in a scope inside the construct
- Objects with dynamic storage duration
- Static data members
- The loop iteration variables in the associated for-loops for a work-sharing **for** or **parallel for** construct
- Static and declared in a scope inside the construct

final (*exp*)
If you specify a `final` clause and *exp* evaluates to a nonzero value, the generated task is a final task. All task constructs encountered inside a final task create final and included tasks.

You can specify only one `final` clause on the **task** pragma.

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.

if (*exp*)
When the `if` clause is specified, an undeferred task is generated if the scalar expression *exp* evaluates to a nonzero value. Only one `if` clause can be specified.

mergeable
If you specify a `mergeable` clause and the generated task is an undeferred task or included task, a merged task might be generated.

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.

shared (*list*)
Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

untied
When a task region is suspended, untied tasks can be resumed by any thread in a team. The `untied` clause on a task construct is ignored if either of the following conditions is a nonzero value:

- A `final` clause is specified on the same task construct and the `final` clause expression evaluates to a nonzero value.
- The task is an included task.

Usage

A final task is a task that makes all its child tasks become final and included tasks. A final task is generated when either of the following conditions is a nonzero value:

- A `final` clause is specified on a task construct and the `final` clause expression evaluates to nonzero value.
- The generated task is a child task of a final task.

An undeferred task is a task whose execution is not deferred with respect to its generating task region. In other words, the generating task region is suspended until the undeferred task has finished running. An undeferred task is generated when an `if` clause is specified on a task construct and the `if` clause expression evaluates to zero.

An included task is a task whose execution is sequentially included in the generating task region. In other words, an included task is undeferred and executed immediately by the encountering thread. An included task is generated when the generated task is a child task of a final task.

A merged task is a task that has the same data environment as that of its generating task region. A merged task might be generated when both the following conditions nonzero values:

- A `mergeable` clause is specified on a task construct.
- The generated task is an undeferred task or an included task.

The `if` clause expression and the `final` clause expression are evaluated outside of the task construct, and the evaluation order is not specified.

Related reference:
"#pragma omp taskwait"

#pragma omp taskyield
Purpose

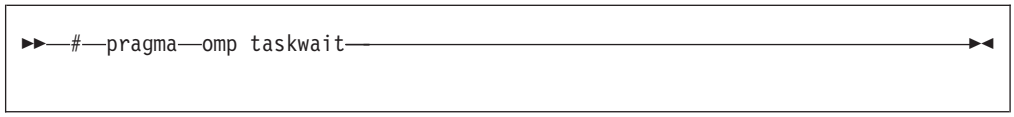
The **omp taskyield** pragma instructs the compiler to suspend the current task in favor of running a different task. The **taskyield** region includes an explicit task scheduling point in the current task region.

Syntax

#pragma omp taskwait
Purpose

Use the **taskwait** pragma to specify a *wait* for child tasks to be completed that are generated by the current task.

Syntax

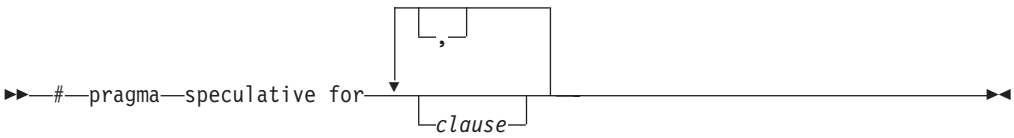


Related reference:
"#pragma omp task" on page 373

#pragma speculative for

The **speculative for** directive instructs the compiler to speculatively parallelize a `for` loop.

Syntax



clause is any of the following clauses:

`default (shared | none)`
Defines the default data scope of variables in each thread.

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 following exceptions:

- The variables are const-qualified.

- The variables are specified in an enclosed data scope attribute clause.
- The variables are used as a loop control variable referenced only by a corresponding **speculative for** directive.

shared (*list***)**
Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

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.

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, is the value assigned to that variable in the last iteration. Variables not assigned a value have an indeterminate value. 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.

reduction (*operator***: *list*)**
Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas. For details, see **#pragma omp for**.

schedule (*type***)**
Specifies how iterations of the `for` loop are divided among available threads. Thread-level speculative execution supports static scheduling. Acceptable values for *type* are as follows:

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*.

Note: You can also specify **dynamic**, **dynamic,*n***, **guided**, **guided,*n*** for *type*, but the runtime treads all these values as static scheduling.

Usage

The directives for thread-level speculative execution only take effect if you specify the -qsmp=speculative compiler option.

The **speculative for** directive must immediately precede a `for` loop.

You cannot create a local object of variable length array within a thread-level speculative execution region.

Example

```
int p=0;
int i=1;

#pragma speculative for firstprivate(p)
for (i=1; i<10; i++)
{
   p++;
}
```

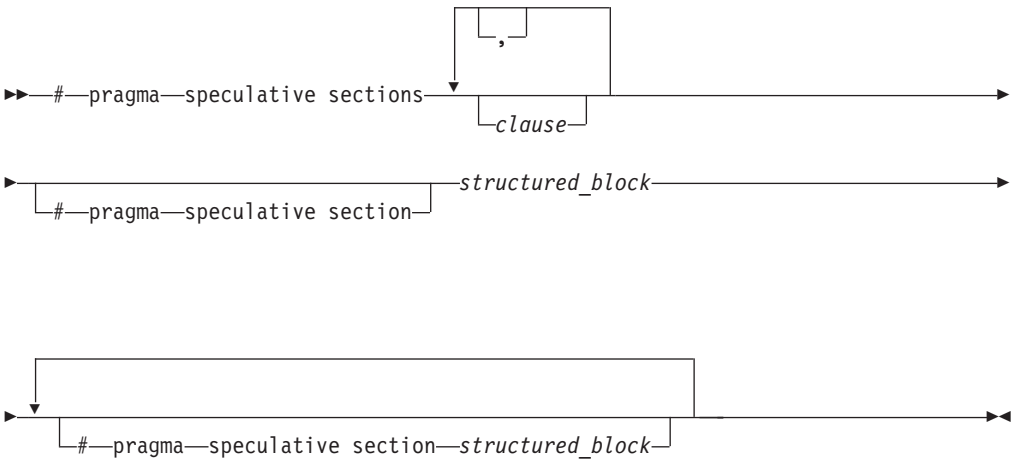
Related information

- The "-qsmp" compiler option
- Thread-level speculative execution
- #pragma speculative sections
- Built-in functions for thread-level speculative execution
- Environment variables for thread-level speculative execution

#pragma speculative section, #pragma speculative sections

The **speculative sections** directive instructs the compiler to speculatively parallelize sections of the code. In code blocks delimited by **speculative sections**, you can use the **speculative section** directive to delimit program code segments.

Syntax



clause is any of the following clauses:

default (shared | none)
Defines the default data scope of variables in each thread.

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 following exceptions:

- The variables are const-qualified.
- The variables are specified in an enclosed data scope attribute clause.
- The variables are used as a loop control variable referenced only by a corresponding **speculative sections** directive.

shared (*list***)**
Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

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.

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, is the value assigned to that variable in the last iteration. Variables not assigned a value have an indeterminate value. 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.

reduction (*operator***: *list*)**
Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas. For details, see **#pragma omp for**.

Usage

The directives for thread-level speculative execution only take effect if you specify the "-qsmp=speculative" compiler option.

The **speculative section** directive is optional for the first code segment inside the **speculative sections** directive. The code segments after the first one must be preceded by a **speculative section** directive. You must use all the **speculative section** directives only in the lexical construct of the code segment that is associated with the **speculative sections** directive.

You cannot create a local object of variable length array within a thread-level speculative execution region.

Example

```
int p=0;
int i=1;

#pragma speculative sections firstprivate(p)
{
    #pragma speculative section
    {
        p++;
    }
    #pragma speculative section
    {
        i--;
    }
}
```

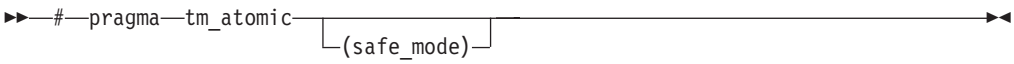
Related information

- The "-qsmp" compiler option
- Thread-level speculative execution
- #pragma speculative for
- Built-in functions for thread-level speculative execution
- Environment variables for thread-level speculative execution

#pragma tm_atomic

The **tm_atomic** directive indicates a transactional atomic region.

Syntax



safe_mode
Using the **safe_mode** clause reduces overhead and increases performance. However, if **safe_mode** is specified, irrevocable actions are not checked at runtime. The run result is undefined if an irrevocable action occurs during the execution.

Usage

The transactional memory directive is enabled with the -qtm compiler option. To compile your program with transactional memory, you must use thread safe invocation commands.

This directive must be placed immediately before the code block of the transactional atomic region.

A transactional atomic region must be one of the following structured blocks:

- An executable statement, possibly compound, with a single entry at the top and a single exit at the bottom
- An OpenMP construct

You cannot branch into or break out from the middle of a transactional atomic region. For example, you cannot use exception raising statements, call `setjmp()`, `longjmp()`, `exit()`, or use the `go to` statement in the middle of a transactional atomic region.

You cannot create a local object of variable length array in a transactional atomic region.

Transactional atomic regions can be nested and the maximum nesting level is 2^{22}. However, the atomicity, consistency, and isolation properties are provided at the outermost transactional atomic region.

- Stopping an inner transaction causes the corresponding outer transaction to stop.
- An inner transaction does not commit data at the end of its transactional atomic region. Instead, the data of the inner transaction is committed later when the data of the corresponding outer transaction is committed.
- If a conflict occurs in a nested transaction, the thread is rolled back to the beginning of the outermost transaction.

Example

```
int v, w, z;
int a[10], b[10];

#pragma omp parallel for
  for (v = 0; v < 10; v++)
    for (w = 0; w < 10; w++)
      for (z = 0; z < 10; z++)
      {
        //Use the tm_atomic directive to indicate a transactional atomic region.
        #pragma tm_atomic
        {
          a[v] = a[w] + b[z];
        }
      }
```

Related information

- The "-qtm" compiler option
- Execution modes
- Built-in functions for transactional memory
- Environment variables for transactional memory
- "Exception handling (C++ only)"

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 383
- "Macros related to compiler features" on page 384

"Examples of predefined macros" on page 393 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 40. 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 `size_t` on the current platform. Not protected.	`unsigned long` in 64-bit compilation mode.
__TIME__	Indicates the time that the source file was preprocessed.	A character string containing the time when the source file was preprocessed.

Table 40. General predefined macros (continued)

Predefined macro name	Description	Predefined value
__TIMESTAMP__	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.	A character string literal in the form "*Day Mmm dd hh:mm:ss yyyy*", where:: *Day* Represents the day of the week (`Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`, or `Sun`). *Mmm* Represents the month in an abbreviated form (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`). *dd* Represents the day. If the day is less than 10, the first `d` is a blank character. *hh* Represents the hour. *mm* Represents the minutes. *ss* Represents the seconds. *yyyy* Represents the year.

Macros indicating the XL C/C++ compiler product

Macros related to the XL C/C++ compiler are always predefined, and are protected (the compiler issues a warning if you try to undefine or redefine them). You can use the **-qshowmacros=pre -E** compiler options to view the values of the predefined macros.

Table 41. Compiler product predefined macros

Predefined macro name	Description	Predefined value
C __IBMC__	Indicates the level of the XL C compiler.	An integer in the format *VRM*, where : *V* Represents the version number *R* Represents the release number *M* Represents the modification number
C++ __IBMCPP__	Indicates the level of the XL C++ compiler.	An integer in the format *VRM*, where : *V* Represents the version number *R* Represents the release number *M* Represents the modification number
C __xlc__	Indicates the level of the XL C compiler.	A string in the format "*V.R.M.F*", where: *V* Represents the version number *R* Represents the release number *M* Represents the modification number *F* Represents the fix level
__xlC__	Indicates the VR level of the XL C and XL C++ compilers in hexadecimal format. Using the XL C compiler also automatically defines this macro.	A four-digit hexadecimal integer in the format 0x*VVRR*, where: *V* Represents the version number *R* Represents the release number

Table 41. Compiler product predefined macros (continued)

Predefined macro name	Description	Predefined value
__xlC_ver__	Indicates the MF level of the XL C and XL C++ compilers in hexadecimal format. Using the XL C compiler also automatically defines this macro.	A eight-digit hexadecimal integer in the format 0x*0000MMFF*, where: *M* Represents the modification number *F* Represents the fix level For example, in XL C/C++ V12.1, PTF 10.1.0.3, the value of the macro is 0x00000003.

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 can be undefined or redefined without warning unless otherwise specified.

Table 42. Platform-related predefined macros

Predefined macro name	Description	Predefined value	Predefined under the following conditions
__bg__	Indicates that this is a Blue Gene/Q platform.	1	Always predefined for all Blue Gene/Q platforms.
__bgq__	Indicates that the architecture is the processor of Blue Gene/Q.	1	Predefined when the architecture is the processor of Blue Gene/Q.
_BIG_ENDIAN, __BIG_ENDIAN__	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.
__ELF__	Indicates that the ELF object model is in effect.	1	Always predefined for the Blue Gene/Q platform.
C++ __GXX_WEAK__	Indicates that weak symbols are supported (used for template instantiation by the linker).	1	Always predefined.
__powerpc, __powerpc__	Indicates that the target is a Power architecture.	1	Predefined when the target is a Power architecture.
__PPC, __PPC__	Indicates that the target is a Power architecture.	1	Predefined when the target is a Power architecture.
__THW_BLUEGENE, __THW_BLUEGENE__	Indicates that the target architecture is Blue Gene.	1	Predefined when the target is Blue Gene.
__TOS_BGQ__	Indicates that the target architecture is the processor of Blue Gene/Q.	1	Predefined when the target is the processor of Blue Gene/Q.
__unix, __unix__	Indicates that the operating system is a variety of UNIX.	1	Always predefined.
__VECTOR4DOUBLE__	Indicates the support of vector data types on Blue Gene/Q	1	Predefined on Blue Gene/Q

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 387
- "Macros related to language levels" on page 387

Macros related to compiler option settings

The following macros can be tested for various features, including source input characteristics, output file characteristics, and optimization. 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 43. General option-related predefined macros

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
__64BIT__	Indicates that 64-bit compilation mode is in effect.	1	-q64
_CHAR_SIGNED, __CHAR_SIGNED__	Indicates that the default character type is `signed char`.	1	-qchars=signed
_CHAR_UNSIGNED, __CHAR_UNSIGNED__	Indicates that the default character type is `unsigned char`.	1	-qchars=unsigned
C++ __EXCEPTIONS	Indicates that C++ exception handling is enabled.	1	-qeh

Table 43. 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	C **-qasm=gcc** and **-qlanglvl=extc99** | **extc89** | **extended** or **-qkeyword=asm** C++ **-qasm=gcc** and **-qlanglvl=extended**
		0	C **-qnoasm** and **-qlanglvl=extc99** | **extc89** | **extended** or **-qkeyword=asm** C++ **-qnoasm** and **-qlanglvl=extended**
C++ __IBM_STDCPP_ASM	Indicates that support for GCC inline `asm` statements is disabled.	0	**-qnoasm=stdcpp**
__IBM_THREAD_LEVEL_SPECULATIVE_EXECUTION__	Indicates support for thread-level speculative execution.	1	-qsmp=speculative
__IBM_TRANSACTIONAL_MEMORY__	Indicates support for transactional memory.	1	-qtm
C++ __IGNERRNO__	Indicates that system calls do not modify `errno`, thereby enabling certain compiler optimizations.	1	-qignerrno
C++ __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*

Table 43. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
C++ __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*
C++ __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
__LONGDOUBLE64	Indicates that the size of a `long double` type is 64 bits.	1	**-qnoldbl128**
__LONGDOUBLE128, __LONG_DOUBLE_128__	Indicates that the size of a `long double` type is 128 bits.	1	-qldbl128
__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
C++ __RTTI_DYNAMIC_CAST__	Indicates that runtime type identification information for the `dynamic_cast` operator is generated.	1	-qrtti

Table 43. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
C++ __RTTI_TYPE_INFO__	Indicates that runtime type identification information for the `typeid` operator is generated.	1	-qrtti
C++ __NO_RTTI__	Indicates that runtime type identification information is disabled.	1	-qnortti
C++ __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	-qarch=qp

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 44. **-qarch**-related macros*

Macro name	Description	Predefined by the following -qarch suboptions
_ARCH_QP	Indicates that the application is targeted to run on the Blue Gene/Q platform.	qp

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 45. Predefined macros for language features

Predefined macro name	Description	Predefined when the following language level is in effect
C++ __BOOL__	Indicates that the `bool` keyword is accepted.	Always defined except when **-qnokeyword=bool** is in effect.
C __C99_BOOL	Indicates support for the `_Bool` data type.	extc1x | stdc99 | extc99 | extc89 | extended
C++ __C99_COMPLEX	Indicates that the support for C99 complex types is enabled or that the C99 complex header should be included.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_CPLUSCMT	Indicates support for C++ style comments	extc1x | stdc99 | extc99 | stdc89 | extc89 | extended (also **-qcpluscmt**)
__C99_COMPOUND_LITERAL	Indicates support for compound literals.	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended | extended0x
C __C99_DESIGNATED_INITIALIZER	Indicates support for designated initialization.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_DUP_TYPE_QUALIFIER	Indicates support for duplicated type qualifiers.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_EMPTY_MACRO_ARGUMENTS	Indicates support for empty macro arguments.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_FLEXIBLE_ARRAY_MEMBER	Indicates support for flexible array members.	extc1x | stdc99 | extc99 | extc89 | extended
__C99__FUNC__	Indicates support for the `__func__` predefined identifier.	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended | extended0x | c99__func__
__C99_HEX_FLOAT_CONST	Indicates support for hexadecimal floating constants.	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended | extended0x | c99hexfloat
C __C99_INLINE	Indicates support for the `inline` function specifier.	extc1x | stdc99 | extc99 (also **-qkeyword=inline**)
__C99_LLONG	Indicates support for C99-style `long long` data types and literals.	C extc1x | stdc99 | extc99 C++ extended0x | c99longlong
__C99_MACRO_WITH_VA_ARGS	Indicates support for function-like macros with variable arguments.	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended | extended0x | varargmacros

Table 45. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__C99_MAX_LINE_NUMBER	Indicates that the maximum line number is 2147483647.	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended0x | c99preprocessor
C __C99_MIXED_DECL_AND_CODE	Indicates support for mixed declaration and code.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_MIXED_STRING_CONCAT	Indicates support for concatenation of wide string and non-wide string literals.	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended0x | c99preprocessor
C __C99_NON_LVALUE_ARRAY_SUB	Indicates support for non-lvalue subscripts for arrays.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_NON_CONST_AGGR_INITIALIZER	Indicates support for non-constant aggregate initializers.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_PRAGMA_OPERATOR	Indicates support for the `_Pragma` operator.	C extc1x | stdc99 | extc99 | extc89 | extended C++ extended extended0x
C __C99_REQUIRE_FUNC_DECL	Indicates that implicit function declaration is not supported.	stdc99
__C99_RESTRICT	Indicates support for the C99 `restrict` qualifier.	C extc1x | stdc99 | extc99 (also **-qkeyword=restrict**) C++ extended | extended0x (also **-qkeyword=restrict**)
C __C99_STATIC_ARRAY_SIZE	Indicates support for the `static` keyword in array parameters to functions.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_STD_PRAGMAS	Indicates support for standard pragmas.	extc1x | stdc99 | extc99 | extc89 | extended
C __C99_TGMATH	Indicates support for type-generic macros in tgmath.h	extc1x | stdc99 | extc99 | extc89 | extended
__C99_UCN	Indicates support for universal character names.	C extc1x | stdc99 | extc99 | ucs C++ ucs
C __C99_VAR_LEN_ARRAY	Indicates support for variable length arrays.	extc1x | stdc99 | extc99 | extc89 | extended
C++ __C99_VARIABLE_LENGTH_ARRAY	Indicates support for variable length arrays.	extended | extended0x | c99vla

Table 45. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__DIGRAPHS__	Indicates support for digraphs.	C extc1x | stdc99 | extc99 | extc89 | extended (also **-qdigraph**) C++ extended | extended0x | compat366 | strict98(also **-qdigraph**)
C __EXTENDED__	Indicates that language extensions are supported.	extended
C++ __IBM__ALIGN	Indicates support for the `__align` specifier.	Always defined except when **-qnokeyword=__alignof** is specified
__IBM_ALIGNOF__, __IBM__ALIGNOF__	Indicates support for the `__alignof__` operator.	C extc1x | extc99 | extc89 | extended C++ extended
__IBM_ATTRIBUTES	Indicates support for type, variable, and function attributes.	C extc1x | extc99 | extc89 | extended C++ extended | extended0x
__IBM_COMPUTED_GOTO	Indicates support for computed `goto` statements.	C extc1x | extc99 | extc89 | extended C++ extended | extended0x | gnu_computedgoto
__IBM_EXTENSION_KEYWORD	Indicates support for the `__extension__` keyword.	C extc1x | extc99 | extc89 | extended C++ extended | extended0x | compat366 | strict98
C __IBM_GCC__INLINE__	Indicates support for the GCC `__inline__` specifier.	extc1x | extc99 | extc89 | extended
C __IBM_DOLLAR_IN_ID	Indicates support for dollar signs in identifiers.	extc1x | extc99 | extc89 | extended
C __IBM_GENERALIZED_LVALUE	Indicates support for generalized lvalues.	extc1x | extc99 | extc89 | extended
__IBM_INCLUDE_NEXT	Indicates support for the `#include_next` preprocessing directive.	C Always defined C++ Always defined except when **-qlanglvl=nognu_include_next** is in effect.

Table 45. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__IBM_LABEL_VALUE	Indicates support for labels as values.	C extc1x | extc99 | extc89 | extended C++ extended | extended0x |gnu_labelvalue
__IBM_LOCAL_LABEL	Indicates support for local labels.	C extc1x | extc99 | extc89 | extended C++ extended | extended0x | gnu_locallabel
__IBM_MACRO_WITH_VA_ARGS	Indicates support for variadic macro extensions.	C extc1x | extc99 | extc89 | extended C++ extended | extended0x | gnu_varargmacros
C _IBM_NESTED_FUNCTION	Indicates support for nested functions.	extc1x | extc99 | extc89 | extended
C __IBM_PP_PREDICATE	Indicates support for `#assert`, `#unassert`, `#cpu`, `#machine`, and `#system` preprocessing directives.	extc1x | extc99 | extc89 | extended
C __IBM_PP_WARNING	Indicates support for the `#warning` preprocessing directive.	extc1x | extc99 | extc89 | extended
C __IBM_REGISTER_VARS	Indicates support for variables in specified registers.	Always defined.
__IBM__TYPEOF__	Indicates support for the `__typeof__ or typeof` keyword.	C always defined C++ extended | extended0x (Also **-qkeyword=typeof**)
__IBMC_COMPLEX_INIT	Indicates support for the initialization of complex types: float _Complex, double _Complex, and long double _Complex.	extc1x
__IBMC_NORETURN	Indicates support for the `_Noreturn` function specifier.	C extc89 | extc99 | extended | extc1x C++ extended | extended0x | c1xnoreturn
C1X __IBMC_STATIC_ASSERT	Indicates support for the static assertions feature.	extc1x
C++0x __IBMCPP_AUTO_TYPEDEDUCTION	Indicates support for the auto type deduction feature.	extended0x | autotypededuction
C++0x __IBMCPP_C99_LONG_LONG	Indicates support for the C99 `long long` feature.	extended0x | c99longlong

Table 45. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
C++0x __IBMCPP_C99_PREPROCESSOR	Indicates support for the C99 preprocessor features adopted in the C++0x standard.	extended0x | c99preprocessor
__IBMCPP_COMPLEX_INIT	Indicates support for the initialization of complex types: float _Complex, double _Complex, and long double _Complex.	extended
C++0x __IBMCPP_CONSTEXPR	Indicates support for the generalized constant expressions feature. **Note:** In XL C/C++ V12.1, this feature is a partial implementation of what is defined in the C++0x standard.	extended0x | constexpr
C++0x __IBMCPP_DECLTYPE	Indicates support for the decltype feature.	extended0x | decltype
C++0x __IBMCPP_DELEGATING_CTORS	Indicates support for the delegating constructors feature.	extended0x | delegatingctors
C++0x __IBMCPP_EXPLICIT_CONVERSION_OPERATORS	Indicates support for the explicit conversion operators feature.	extended0x | explicitconversionoperators
C++0x __IBMCPP_EXTENDED_FRIEND	Indicates support for the extended friend declarations feature.	extended0x | extendedfriend
C++0x __IBMCPP_EXTERN_TEMPLATE	Indicates support for the explicit instantiation declarations feature.	extended | extended0x | externtemplate
C++0x __IBMCPP_INLINE_NAMESPACE	Indicates support for the inline namespace definitions feature.	extended0x | inlinenamespace
C++0x __IBMCPP_REFERENCE_COLLAPSING	Indicates support for the reference collapsing feature.	extended0x | referencecollapsing
C++0x __IBMCPP_RIGHT_ANGLE_BRACKET	Indicates support for the right angle bracket feature.	extended0x | rightanglebracket
C++0x __IBMCPP_RVALUE_REFERENCES	Indicates support for the rvalue references feature.	extended0x | rvaluereferences
C++0x __IBMCPP_SCOPED_ENUM	Indicates support for the scoped enumeration feature.	extended0x | scopedenum
C++0x __IBMCPP_STATIC_ASSERT	Indicates support for the static assertions feature.	C++ extended0x | static_assert
C++0x __IBMCPP_VARIADIC_TEMPLATES	Indicates support for the variadic templates feature.	extended0x | variadic[templates]

Table 45. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
_LONG_LONG	Indicates support for `long long` data types.	C extc1x | stdc99 | extc99 | | stdc89 | extc89 | extended (also -qlonglong) C++ extended0x | c99longlong | extended (also **-qlonglong**)
C __STDC__	Indicates that the compiler conforms to the ANSI/ISO C standard.	Predefined to 1 if ANSI/ISO C standard conformance is in effect.
__STDC_HOSTED__	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).	C extc1x | stdc99 | extc99 C++ extended0x
C __STDC_VERSION__	Indicates the version of ANSI/ISO C standard which the compiler conforms to.	The format is yyyymmL. (For example, the format is 199901L 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
```

C++ 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. 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 Blue Gene 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.

C++ 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.

The following sections describe the available built-in functions for the Blue Gene/Q platform.

Fixed-point built-in functions

Fixed-point built-in functions are grouped into the following categories:

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

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
Purpose

Load Halfword Byte Reversed, Load Word Byte Reversed

Prototype

```
unsigned short __load2r (unsigned short*);

unsigned int __load4r (unsigned int*);
```

Multiply functions

__mulhd, __mulhdu
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);
```

__mulhw, __mulhwu
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
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
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
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
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
Purpose

Store 2/4-byte Reversal

Prototype

void __store2r (unsigned short, unsigned short*);

void __store4r (unsigned int, unsigned int*);

Trap functions

__tdw, __tw
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

T0 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.

__trap, __trapd
Purpose

Trap if the Parameter is not Zero, Trap if the Parameter is not Zero Doubleword

Prototype

void __trap (int);

void __trapd (long);

Binary floating-point built-in functions

Floating-point built-in functions are grouped into the following categories:

Absolute value functions

__fabss
Purpose

Floating Absolute Value Single

Returns the absolute value of the argument.

Prototype

```
float __fabss (float);
```

__fnabs
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

__cmplx, __cmplxf, __cmplxl
Purpose

Converts two real parameters into a single complex value.

Prototype

```
double _Complex __cmplx (double, double);

float _Complex __cmplxf (float, float);

long double _Complex __cmplxl (long double, long double);
```

__fcfid
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 __fcfid (double);
```

__fctid
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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 408.

__fre, __fres
Purpose

Floating Reciprocal Estimate, Floating Reciprocal Estimate Single

Prototype

```
float __fre (double);

float __fres (float);
```

Usage

`__fre` is valid only when **-qarch** is set to target POWER5 or later processors.

Rounding functions

__frim, __frims
Purpose

Floating Round to Integer Minus

Rounds the floating-point argument to an integer using round-to-minus-infinity mode, and returns the value as a floating-point value.

Prototype

```
double __frim (double);

float __frims (float);
```

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

__frin, __frins
Purpose

Floating Round to Integer Nearest

Rounds the floating-point argument to an integer using round-to-nearest mode, and returns the value as a floating-point value.

Prototype

```
double __frin (double);

float __frins (float);
```

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

__frip, __frips
Purpose

Floating Round to Integer Plus

Rounds the floating-point argument to an integer using round-to-plus-infinity mode, and returns the value as a floating-point value.

Prototype

```
double __frip (double);

float __frips (float);
```

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

__friz, __frizs
Purpose

Floating Round to Integer Zero

Rounds the floating-point argument to an integer using round-to-zero mode, and returns the value as a floating-point value.

Prototype

```
double __friz (double);

float __frizs (float);
```

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

Select functions

__fsel, __fsels
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

__frsqrte, __frsqrtes
Purpose

Floating Reciprocal Square Root Estimate, Floating Reciprocal Square Root Estimate Single

Prototype

```
double __frsqrte (double);

float __frsqrtes (float);
```

Usage

`__frsqrtes` is valid only when **-qarch** is set to target POWER5+ or later processors.

__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 `__swdiv` built-in function in situations where division is performed repeatedly in a loop and when arguments are within the permitted ranges.

Store functions

__stfiw
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

Synchronization and atomic built-in functions are grouped into the following categories:

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.

__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.

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*,

__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*.

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.

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_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.

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`,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 485 for a description of the `__fence` built-in function).

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 `__lwarx`) 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.

Synchronization functions

__eieio, __iospace_eioio
Purpose

Enforce In-order Execution of Input/Output

Ensures that all I/O storage access instructions preceding the call to `__eioeio` 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
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);
```

__lwsync, __iospace_lwsync
Purpose

Load Word Synchronize

Ensures that all instructions preceding the call to `__lwsync` complete before any subsequent store instructions can be executed on the processor that executed the function. Also, it ensures that all load instructions preceding the call to `__lwsync` complete before any subsequent load 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, __iospace_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);

void __iospace_sync (void);
```

Cache-related built-in functions

Cache-related built-in functions are grouped into the following categories:

- "Data cache functions" on page 417
- "Prefetch built-in functions" on page 418

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);
```

__dcbfl
Purpose

Data Cache Block Flush Line

Flushes the cache line at the specified address from the L1 data cache.

Prototype

```
void __dcbfl (const void* addr );
```

Usage

The target storage block is preserved in the L2 cache.

Valid only when **-qarch** is set to target POWER6® processors

__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);
```

__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 built-in 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 consecutive memory locations by using an explicit stream.

Prototype

```
void __prefetch_by_stream (const int, const void*);
```

Block-related built-in functions

__bcopy
Purpose

Copies *n* bytes from *src* to *dest*. The result is correct even when both areas overlap.

Prototype

```
void __bcopy(const void* src, void* dest, size_t n);
```

Parameters

src
The source address of data to be copied.

dest
The destination address of data to be copied

n The size of the data.

Vector built-in functions

Individual elements of vectors can be accessed by using the Quad Processing Extension (QPX) built-in functions. This section provides an alphabetical reference to the QPX built-in functions. You can use these functions to manipulate vectors.

You must specify appropriate compiler options for your architecture when you use the built-in functions.

This section uses pseudocode description to represent function syntax, as shown below:

```
d=func_name(a, b, c)
```

In the description,

- `d` represents the return value of the function.
- `a`, `b`, and `c` represent the arguments of the function.
- `func_name` is the name of the function.

For example, the syntax for the function `vector4double vec_add(vector4double, vector4double);` is represented by `d=vec_add(a, b)`.

Some built-in functions depend on the value of the floating-point status and control register (FPSCR). For information on the FPSCR, see "FPSCR functions" on page 403.

Floating-point operands for logical functions

In the quad vector logical functions, such as `vec_and`, floating-point operands are interpreted in the following ways:

- Any value that is greater than or equal to zero (both positive zero and negative zero) is interpreted as the `true` logical value.
- Any value that is less than zero is interpreted as the `false` logical value.
- NaN is interpreted as false.

In the result values, floating-point boolean values are as follows:

- `true` is 1.0.
- `false` is -1.0.

vec_abs

Purpose

Returns a vector containing the absolute values of the contents of the given vector.

Syntax

```
d=vec_abs(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

The value of each element of the result is the absolute value of the corresponding element of a.

Formula

```
d[0] = |a[0]|
d[1] = |a[1]|
d[2] = |a[2]|
d[3] = |a[3]|
```

Example

```
a =  (10.0, -20.0, 30.0, -40.0)
d: (10.0,  20.0, 30.0,  40.0)
```

vec_add

Purpose

Returns a vector containing the sums of each set of corresponding elements of the given vectors.

Syntax

```
d=vec_add(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is the sum of the corresponding elements of a and b.

Formula

```
d[0] = a[0] + b[0]
d[1] = a[1] + b[1]
d[2] = a[2] + b[2]
d[3] = a[3] + b[3]
```

Example

```
a =  (10.0, 20.0,  30.0,  40.0)
b =  (50.0, 60.0,  70.0,  80.0)
d: (60.0, 80.0, 100.0, 120.0)
```

vec_and

Purpose

Returns a vector containing the results of performing a logical AND operation between the given vectors.

Syntax

```
d=vec_and(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is the result of a logical AND operation between the corresponding elements of a and b.

Formula

```
d[0] = a[0] AND b[0]
d[1] = a[1] AND b[1]
d[2] = a[2] AND b[2]
d[3] = a[3] AND b[3]
```

Example

```
a =  (-1.0, -1.0,  1.0, 1.0)
b =  (-1.0,  1.0, -1.0, 1.0)
d: (-1.0, -1.0, -1.0, 1.0)
```

vec_andc

Purpose

Returns a vector containing the results of performing a logical AND operation between a and the complement of b.

Syntax

```
d=vec_andc(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is the result of a logical AND operation between the corresponding element of a and the complement of the corresponding element of b.

Formula

```
d[0] = a[0] AND NOT (b[0])
d[1] = a[1] AND NOT (b[1])
d[2] = a[2] AND NOT (b[2])
d[3] = a[3] AND NOT (b[3])
```

Example

```
a =  (-1.0, -1.0,  1.0, 1.0)
b =  (-1.0,  1.0, -1.0, 1.0)
d: (-1.0, -1.0,  1.0,-1.0)
```

vec_ceil

Purpose

Returns a vector containing the smallest representable floating-point integral values greater than or equal to the values of the corresponding elements of the given vector.

Syntax

```
d=vec_ceil(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of the result contains the smallest representable floating-point integral value greater than or equal to the value of the corresponding element of a.

Example

```
a  = (-5.8, -2.3, 2.3, 5.8)
d: (-5.0, -2.0, 3.0, 6.0)
```

vec_cmpeq

Purpose

Returns a vector containing the results of comparing each set of corresponding elements of the given vectors for equality.

Syntax

```
d=vec_cmpeq(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is 1.0 if the corresponding element of a is equal to the corresponding element of b. Otherwise, the value is -1.0.

Formula

```
If (a[0] EQ b[0]) Then d[0] = 1.0 Else d[0] = -1.0
If (a[1] EQ b[1]) Then d[1] = 1.0 Else d[1] = -1.0
If (a[2] EQ b[2]) Then d[2] = 1.0 Else d[2] = -1.0
If (a[3] EQ b[3]) Then d[3] = 1.0 Else d[3] = -1.0
```

Note: EQ is the equal operator.

Example

```
a =  (10.0, -10.0, -10.0,  80.0)
b =  (10.0,  20.0, -10.0, -40.0)
d: ( 1.0,  -1.0,   1.0,  -1.0)
```

vec_cmpgt

Purpose

Returns a vector containing the results of a greater-than comparison between each set of corresponding elements of the given vectors.

Syntax

```
d=vec_cmpgt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is 1.0 if the corresponding element of a is greater than the corresponding element of b. Otherwise, the value is -1.0.

Formula

```
If (a[0] > b[0]) Then d[0] = 1.0 Else d[0] = -1.0
If (a[1] > b[1]) Then d[1] = 1.0 Else d[1] = -1.0
If (a[2] > b[2]) Then d[2] = 1.0 Else d[2] = -1.0
If (a[3] > b[3]) Then d[3] = 1.0 Else d[3] = -1.0
```

Example

```
a =  (10.0,  20.0, 30.0, -40.0)
b =  (20.0, -10.0, 10.0,  80.0)
d: (-1.0,   1.0,  1.0,  -1.0)
```

vec_cmplt

Purpose

Returns a vector containing the results of a less-than comparison between each set of corresponding elements of the given vectors.

Syntax

```
d=vec_cmplt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is 1.0 if the corresponding element of `a` is less than the corresponding element of `b`. Otherwise, the value is -1.0.

Formula

```
If (a[0] < b[0]) Then d[0] = 1.0 Else d[0] = -1.0
If (a[1] < b[1]) Then d[1] = 1.0 Else d[1] = -1.0
If (a[2] < b[2]) Then d[2] = 1.0 Else d[2] = -1.0
If (a[3] < b[3]) Then d[3] = 1.0 Else d[3] = -1.0
```

Example

```
a =  (20.0, -10.0, 10.0,  80.0)
b =  (10.0,  20.0, 30.0, -40.0)
d: (-1.0,   1.0,  1.0,  -1.0)
```

vec_cpsgn

Purpose

Returns a vector by copying the sign of the elements in vector `a` to the sign of the corresponding elements in vector `b`.

Syntax

```
d=vec_cpsgn(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The values of the elements of the result are obtained by copying the sign of the elements in a to the sign of the corresponding elements in b.

Formula

```
d[0] = (double) { sign(a[0]), mantissa(b[0]), exponent(b[0]) }
d[1] = (double) { sign(a[1]), mantissa(b[1]), exponent(b[1]) }
d[2] = (double) { sign(a[2]), mantissa(b[2]), exponent(b[2]) }
d[3] = (double) { sign(a[3]), mantissa(b[3]), exponent(b[3]) }
```

Example

```
a =  (   -1.0,    2.0,    -3.0,    4.0)
b =  ( 1.5e10, 2.5e15,  3.5e20, 4.5e25)
d: (-1.5e10, 2.5e15, -3.5e20, 4.5e25)
```

vec_cfid

Purpose

Returns a vector of which each element is the floating point equivalent of the 64-bit signed integer in the corresponding element of a, rounded to double-precision, using the rounding mode specified by $FPSCR_{RN}$.

Syntax

```
d=vec_cfid(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

The value of each element of the result is the floating-point representation of the 64-bit signed integer in the corresponding element of a, rounded to double-precision using the rounding mode specified by $FPSCR_{RN}$.

Example

```
FPSCR[RN] = DFP_ROUND_TO_NEAREST_WITH_TIES_TO_EVEN
a =  (   1,   -1,   2,   -2)
d: ( 1.0, -1.0, 2.0, -2.0)
```

Related functions

- "FPSCR functions" on page 403

vec_cfidu

Purpose

Returns a vector of which each element is the floating point equivalent of the 64-bit unsigned integer in the corresponding element of a, rounded to double-precision, using the rounding mode specified by $FPSCR_{RN}$.

Syntax

```
d=vec_cfidu(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

The value of each element of the result is the floating-point representation of the 64-bit unsigned integer in the corresponding element of a, rounded to double-precision using the rounding mode specified by $FPSCR_{RN}$.

Example

```
FPSCR[RN] = DFP_ROUND_TO_NEAREST_WITH_TIES_TO_EVEN
a =  (   1,   2,   3,   4)
d: ( 1.0, 2.0, 3.0, 4.0)
```

Related functions

- "FPSCR functions" on page 403

vec_ctid

Purpose

Converts a quad vector to 64-bit signed integer values.

Syntax

```
d=vec_ctid(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of a is rounded to floating-point integral value according to $FPSCR_{RN}$. The corresponding element of the result vector is then set to one of the following values:

- If the rounded value is greater than 2^{63}-1, the result is maximal long integer (0x7FFF FFFF FFFF FFFF).
- If the rounded value is less than -2^{63}, the result is minimal long integer (0x8000 0000 0000 0000).
- Otherwise, the result is the 64-bit signed integer value equivalent to the rounded value.

Example

```
FPSCR[RN] = DFP_ROUND_TOWARD_POSITIVE_INFINITY
a =  (1.4, -2.9,               9.0e20,              -5.0e25)
d: (  2,   -2, 0x7FFF FFFF FFFF FFFF, 0x8000 0000 0000 0000)
```

Related functions

- "FPSCR functions" on page 403

vec_ctidu

Purpose

Converts a quad vector to 64-bit unsigned integer values.

Syntax

```
d=vec_ctidu(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of a is rounded to floating-point integral value according to $FPSCR_{RN}$. The corresponding element of the result vector is then set to one of the following values:

- If the rounded value is greater than 2^{64}-1, the result is maximal unsigned long integer (0xFFFF FFFF FFFF FFFF).
- If the rounded value is less than 0, the result is 0 (0x0000 0000 0000 0000).
- Otherwise, the result is the 64-bit unsigned integer value equivalent to the rounded value.

Example

```
FPSCR[RN] = DFP_ROUND_TOWARD_POSITIVE_INFINITY
a =  (1.4, 1.9,                9.0e22, -5.0e25)
d: (  2,   2, 0xFFFF FFFF FFFF FFFF,       0)
```

Related functions

- "FPSCR functions" on page 403

vec_ctiduz

Purpose

Converts a quad vector to 64-bit unsigned integer values with rounding toward zero.

Syntax

```
d=vec_ctiduz(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of a is rounded towards to zero to floating-point integral value. The corresponding element of the result vector is then set to one of the following values:

- If the rounded value is greater than 2^{64}-1, the result is maximal unsigned long integer (0xFFFF FFFF FFFF FFFF).
- If the rounded value is less than 0, the result is 0 (0x0000 0000 0000 0000).
- Otherwise, the result is the 64-bit unsigned integer value equivalent to the rounded value.

Example

```
a =  (1.6, -8.8,                9.0e22, -5.0e25)
d: (  1,    0, 0xFFFF FFFF FFFF FFFF,       0)
```

vec_ctidz

Purpose

Converts a quad vector to 64-bit signed integer values with rounding toward zero.

Syntax

```
d=vec_ctidz(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of a is rounded towards zero to floating-point integral value. The corresponding element of the result vector is then set to one of the following values:

- If the rounded value is greater than 2^{63}-1, the result is maximal long integer (0x7FFF FFFF FFFF FFFF).
- If the rounded value is less than -2^{63}, the result is minimal long integer (0x8000 0000 0000 0000).
- Otherwise, the result is the 64-bit signed integer value equivalent to the rounded value.

Example

```
a =  (1.6, -1.9,                 9.0e20,               -5.0e25)
d: (  1,   -1, 0x7FFF FFFF FFFF FFFF , 0x8000 0000 0000 0000)
```

vec_ctiw

Purpose

Converts a quad vector to 32-bit signed integer values.

Syntax

```
d=vec_ctiw(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of a is rounded to floating-point integral value according to $FPSCR_{RN}$. The four low-order bytes of the corresponding element of the result vector then contain one of the following values:

- If the rounded value is greater than 2^{31}-1, the result is maximal integer (0x7FFF FFFF).
- If the rounded value is less than -2^{31}, the result is minimal integer (0x8000 0000).
- Otherwise, the result is the 32-bit signed integer value equivalent to the rounded value.

Example

```
FPSCR_RN = DFP_ROUND_TOWARD_POSITIVE_INFINITY
a =  (1.4, -2.9,      9.0e11,     -5.0e12)
d: (  2,   -2, 0x7FFF FFFF, 0x8000 0000)
```

Related functions

- "FPSCR functions" on page 403

vec_ctiwu

Purpose

Converts a quad vector to 32-bit unsigned integer values.

Syntax

```
d=vec_ctiwu(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of a is rounded to floating-point integral value according to $FPSCR_{RN}$. The four low-order bytes of the corresponding element of the result vector then contain one of the following values:

- If the rounded value is greater than 2^{32}-1, the result is maximal unsigned integer (0xFFFF FFFF).
- If the rounded value is less than 0, the result is 0 (0x0000 0000).
- Otherwise, the result is the 32-bit unsigned integer value equivalent to the rounded value.

Example

```
FPSCR_RN = DFP_ROUND_TOWARD_POSITIVE_INFINITY
a =  (1.4, 1.9,      9.0e11, -5.0e12)
d: (  2,   2, 0xFFFF FFFF,       0)
```

Related functions

- "FPSCR functions" on page 403

vec_ctiwuz

Purpose

Converts a quad vector to 32-bit unsigned integer values with rounding toward zero.

Syntax

```
d=vec_ctiwuz(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of a is rounded towards zero to floating-point integral value. The four low-order bytes of the corresponding element of the result vector then contain one of the following values:

- If the rounded value is greater than 2^{32}-1, the result is maximal unsigned integer (0xFFFF FFFF).
- If the rounded value is less than 0, the result is 0 (0x0000 0000).
- Otherwise, the result is the 32-bit unsigned integer value equivalent to the rounded value.

Example

```
a =  (1.6, -1.9,      9.0e11, -5.0e12)
d: (  1,    0, 0xFFFF FFFF,       0)
```

vec_ctiwz

Purpose

Converts a quad vector to 32-bit signed integer values with rounding toward zero.

Syntax

```
d=vec_ctiwz(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of a is rounded towards zero to floating-point integral value. The four low-order bytes of the corresponding element of the result vector then contain one of the following values:

- If the rounded value is greater than 2^{31}-1, the result is maximal integer (0x7FFF FFFF).
- If the rounded value is less than -2^{31}, the result is minimal integer (0x8000 0000).
- Otherwise, the result is the 32-bit signed integer value equivalent to the rounded value.

Example

```
a =  (1.6, -1.9,      9.0e11,     -5.0e12)
d: (  1,   -1, 0x7FFF FFFF, 0x8000 0000)
```

vec_extract

Purpose

Returns the value of element a from the vector b.

Syntax

```
d=vec_extract(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
double	vector4double	int

Result value

This function uses the modulo arithmetic on b to determine the element number. For example, if b is out of range, the compiler uses b modulo the number of elements in the vector to determine the element position.

Formula

```
d = a[b MOD 4]
```

Note: MOD is the modulo operator.

Example

```
a = (10.0, 20.0, 30.0, 40.0)
b = 1
d: 20.0
```

vec_floor

Purpose

Returns a vector containing the largest representable floating-point integral values less than or equal to the values of the corresponding elements of the given vector.

Syntax

```
d=vec_floor(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of the result contains the largest representable floating-point integral value less than or equal to the value of the corresponding element of a.

Example

```
a  = (-5.8, -2.3, 2.3, 5.8)
d: (-6.0, -3.0, 2.0, 5.0)
```

vec_gpci

Purpose

Returns a vector containing the results of dispersing the 12-bit literal a to be used as control value for a permute instruction.

Note: In this information, constants beginning with 0 are interpreted as octal constants.

Syntax

```
d=vec_gpci(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	int, a value in 00 - 07777

Restriction: You cannot use an integer variable for a.

Result value

The value of each element of the result has a sign bit set to 0, an exponent set to 02000, and a mantissa where bits 0:2 are taken from the 12-bit literal a as shown in the formula.

Formula

$d[0] = (double)\ \{sign = 0, mantissa_{0:2} = a_{0:2}, exponent = 02000\}$
$d[1] = (double)\ \{sign = 0, mantissa_{0:2} = a_{3:5}, exponent = 02000\}$
$d[2] = (double)\ \{sign = 0, mantissa_{0:2} = a_{6:8}, exponent = 02000\}$
$d[3] = (double)\ \{sign = 0, mantissa_{0:2} = a_{9:11}, exponent = 02000\}$

Example

Shifting the elements of a given vector to the left by one step and rotate around requires the pattern 1–2–3–0. It can be obtained by the following code:

```
pattern = vec_gpci(01230);
v = vec_perm(v,v,pattern);
```

With the pattern 1–2–3–0, the vector
(0.0, 1.0, 2.0, 3.0)
becomes
(1.0, 2.0, 3.0, 0.0).

vec_insert

Purpose

Returns a copy of the vector b with the value of its element c replaced by a.

Syntax

```
d=vec_insert(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	double	vector4double	int

Result value

This function uses the modulo arithmetic on c to determine the element number. For example, if c is out of range, the compiler uses c modulo the number of elements in the vector to determine the element position.

Formula

```
If ((c MOD 4) EQ 0) Then d[0] = a Else d[0] = b[0]
If ((c MOD 4) EQ 1) Then d[1] = a Else d[1] = b[1]
If ((c MOD 4) EQ 2) Then d[2] = a Else d[2] = b[2]
If ((c MOD 4) EQ 3) Then d[3] = a Else d[3] = b[3]
```

Notes:

- MOD is the modulo operator.
- EQ is the equal operator.

Example

```
a = 50.0
b = (10.0, 20.0, 30.0, 40.0)
c = 1
d: (10.0, 50.0, 30.0, 40.0)
```

vec_ld, vec_lda

Purpose

Loads a vector from the given memory address.

Syntax

```
d=vec_ld(a, b)
d=vec_lda(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	long	long*
		unsigned long*
		long long*
		unsigned long long*
		float*
		_Complex float*
		double*
		_Complex double*

Result value

The effective address (EA) is the sum of a and b. The effective address is truncated to an *n*-byte alignment depending on the type of b as shown in the following table. The result is the content of the *n* bytes of memory starting at the effective address.

Type of b	*n*
long* unsigned long* long long* unsigned long long*	32
float*	16
_Complex float*	
double*	32
_Complex double*	

vec_lda generates an exception (SIGBUS) if the effective address is not aligned to the appropriate memory boundary indicated in the table.

If b is a pointer to a variable of the single-precision floating-point type or single-precision complex type, the values loaded from memory are converted to double precision before being saved to the result value.

Formula

The following table shows the formulas depending on the type of b.

Type of b	Formula
long* unsigned long* long long* unsigned long long*	d[0]=Memory[EA] d[1]=Memory[EA+8] d[2]=Memory[EA+16] d[3]=Memory[EA+24]
float*	d[0]=(double) Memory_SP[EA] d[1]=(double) Memory_SP[EA+4] d[2]=(double) Memory_SP[EA+8] d[3]=(double) Memory_SP[EA+12]
_Complex float*	
double*	d[0]=Memory[EA] d[1]=Memory[EA+8] d[2]=Memory[EA+16] d[3]=Memory[EA+24]
_Complex double*	

Note: `Memory_SP[]` is a single-precision floating-point array.

Example

Type of b	Memory values	d
long* unsigned long* long long* unsigned long long*	0x4024000000000000, 0x4034000000000000, 0x403E000000000000, 0x4044000000000000	(10.0, 20.0, 30.0, 40.0)
float*	10.0f, 20.0f, 30.0f, 40.0f	(10.0, 20.0, 30.0, 40.0)
_Complex float*	(10.0f, 20.0f) (30.0f, 40.0f)	
double*	10.0, 20.0, 30.0, 40.0	
_Complex double*	(10.0, 20.0) (30.0, 40.0)	

vec_ld2, vec_ld2a

Purpose

Loads a vector from two floating-point values at a given memory address.

Syntax

```
d=vec_ld2(a, b)
d=vec_ld2a(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	long	double*
		float*

Result value

The effective address (EA) is the sum of `a` and `b`. The effective address is truncated to an *n*-byte alignment depending on the type of `b` as shown in the following table. *n* bytes of memory are loaded from memory starting at the effective address and replicated to fill the result.

	Type of b	
	double*	**float***
n	16	8

`vec_ld2a` generates an exception (SIGBUS) if the effective address is not aligned to the appropriate memory boundary indicated in the table.

If `b` is a pointer to a variable of the single-precision floating-point type, the values loaded from memory are converted to double precision before being saved to the result value.

Formula

The following table shows the formulas depending on the type of `b`.

	Type of b	
	double*	**float***
d[0]	Memory[EA]	(double) Memory_SP[EA]
d[1]	Memory[EA+8]	(double) Memory_SP[EA+4]
d[2]	Memory[EA]	(double) Memory_SP[EA]
d[3]	Memory[EA+8]	(double) Memory_SP[EA+4]

Note: `Memory_SP[]` is a single-precision floating-point array.

Example

	Type of b	
	double*	**float***
Memory values	10.0, 20.0	10.0f, 20.0f
d	(10.0, 20.0, 10.0, 20.0)	

vec_ldia, vec_ldiaa

Purpose

Loads a vector from four 4-byte signed integer values at the given memory address, with sign extension to 8-byte signed integer values.

Syntax

```
d=vec_ldia(a, b)
d=vec_ldiaa(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	long	int*

Result value

The effective address (EA) is the sum of `a` and `b`. The effective address is truncated to a 16-byte alignment. The contents of the 16 bytes starting at the effective address are loaded from memory. They are then converted from four 4-byte signed integer values to four 8-byte signed integer values before being saved in the result value.

`vec_ldiaa` generates an exception (SIGBUS) if the effective address is not aligned to a 16-byte memory boundary.

Formula

```
d[0] = (long) Memory_4B[EA]
d[1] = (long) Memory_4B[EA+4]
d[2] = (long) Memory_4B[EA+8]
d[3] = (long) Memory_4B[EA+12]
```

Note: `Memory_4B[]` is a 4-byte signed integer array.

Example

Memory values: `(10, -20, 30, -40)`
Convert result values `d` to IEEE floating point numbers using: `d2 = vec_cfid(d)`
`d2: (10.0, -20.0, 30.0, -40.0)`

vec_ldiz, vec_ldiza

Purpose

Loads a vector from four 4-byte unsigned integer values at the given memory address, with zero extension to 8-byte unsigned integer values.

Syntax

```
d=vec_ldiz(a, b)
d=vec_ldiza(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	long	unsigned*

Result value

The effective address (EA) is the sum of `a` and `b`. The effective address is truncated to a 16-byte alignment. The contents of the 16 bytes starting at the effective address are loaded from memory. Each of their four 4-byte integer values is extended with zeros to fill 8-byte integer values before being saved in the result value.

`vec_ldiza` generates an exception (SIGBUS) if the effective address is not aligned to a 16-byte memory boundary.

Formula

```
d[0] = (unsigned long) Memory_4B[EA]
d[1] = (unsigned long) Memory_4B[EA+4]
d[2] = (unsigned long) Memory_4B[EA+8]
d[3] = (unsigned long) Memory_4B[EA+12]
```

Note: `Memory_4B[]` is a 4-byte integer array.

Example

Memory values: `(10, 20, 30, 40)`
Convert result values `d` to IEEE floating point numbers using: `d2 = vec_cfid(d)`
`d2: (10.0, 20.0, 30.0, 40.0)`

vec_lds, vec_ldsa

Purpose

Loads a vector from a single floating-point or complex value at the given memory address.

Syntax

```
d=vec_lds(a, b)
d=vec_ldsa(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	**a**	**b**
`vector4double`	`long`	`double*` (only for `vec_lds`)
		`float*` (only for `vec_lds`)
		`_Complex double*`
		`_Complex float*`

Result value

The effective address (EA) is the sum of `a` and `b`. If `b` is a pointer to a complex value, the effective address is truncated to an *n*-byte alignment depending on the type of `b` as shown in the following table. The loaded value or complex value is replicated to fill the result.

	Type of b	
	_Complex double*	**_Complex float***
n	16	8

`vec_ldsa` generates an exception (SIGBUS) if the effective address is not aligned to the appropriate memory boundary indicated in the table.

If `b` is a pointer to a variable of the single-precision floating-point type or single-precision complex type, the values loaded from memory are converted to double precision before being saved to the result value.

Formula

The following table shows the formulas depending on the type of b.

	Type of b			
	double*	**float***	**_Complex double***	**_Complex float***
d[0]	Memory[EA]	(double) Memory_SP[EA]	Memory[EA]	(double) Memory_SP[EA]
d[1]	Memory[EA]	(double) Memory_SP[EA]	Memory[EA+8]	(double) Memory_SP[EA+4]
d[2]	Memory[EA]	(double) Memory_SP[EA]	Memory[EA]	(double) Memory_SP[EA]
d[3]	Memory[EA]	(double) Memory_SP[EA]	Memory[EA+8]	(double) Memory_SP[EA+4]

Note: Memory_SP[] is a single-precision floating-point array.

Example

	Type of b			
	double*	**float***	**_Complex double***	**_Complex float***
Memory values	10.0	10.0f	(10.0, 20.0)	(10.0f, 20.0f)
d	(10.0, 10.0, 10.0, 10.0)		(10.0, 20.0, 10.0, 20.0)	

vec_logical

Purpose

Returns a vector containing the results of performing a logical operation between a and b, using the truth table specified by c.

Syntax

```
d=vec_logical(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	int, a value in the range of [0x0, 0xF]

Result value

The value of each element of the result is the result of the logical operation between the corresponding elements of a and b, using the truth table specified by c.

The following table shows how to read the truth table in c for the n^{th} element of a and b.

a[n]	b[n]	Binary result
False	False	c_0
True	False	c_1
False	True	c_2
True	True	c_3

The result value is calculated from the binary result.

Binary result	Result value
0	1.0 (True)
1	-1.0 (False)

Formula

```
If (a[n] < 0.0) AND (b[n] < 0.0)
  If (c0 EQ 0), d[n]= -1.0
  Else d[n]= 1.0
If (a[n] ≥ 0.0) AND (b[n] < 0.0)
  If (c1 EQ 0), d[n]= -1.0
  Else d[n]= 1.0
If (a[n] < 0.0) AND (b[n] ≥ 0.0)
  If (c2 EQ 0), d[n]= -1.0
  Else d[n]= 1.0
If (a[n] ≥ 0.0) AND (b[n] ≥ 0.0)
  If (c3 EQ 0), d[n]= -1.0
  Else d[n]= 1.0
```

Notes:

- EQ is the equal operator.
- In this function, NaN is considered to be less than zero.

Example

You can use the values for c from the following table to replicate some usual logical operators.

Binary	c	Operator
0001	0x1	AND
0110	0x6	XOR
0111	0x7	OR
1000	0x8	NOR
1110	0xE	NAND

vec_lvsl

Purpose

Returns a vector useful for aligning non-aligned data.

Syntax

```
d=vec_lvsl(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	long	double*
		_Complex double*
		float*
		_Complex float*

Result value

The result value is a quad vector. The elements of the quad vector are generated in the following ways:

- Sign: 0
- Mantissa:
  1. For the first element, the mantissa is the result of following operations:
     - If b is a pointer to a double-precision floating-point value or complex value:
       a. Add a and b.
       b. Mask the result of the previous step with 0b11000.
       c. Take the integer value of bits 58 - 60 from the result of the previous step.
     - If b is a pointer to a single-precision floating-point value or complex value:
       a. Add a and b.
       b. Multiply the result of the previous step by two.
       c. Mask the result of the previous step with 0b11000.
       d. Take the integer value of bits 58 - 60 from the result of the previous step.
  2. The mantissa is incremented by one for each subsequent element.

     The mantissa is seen as a 3-bit value for the increment operation. That is, incrementing 0b111 produces 0b000.
- Exponent: 0x400

You can use the result as an argument of the vec_perm function.

Formula

The following formula is applicable if b is a pointer to a double-precision floating-point value or complex value:

```
EA = a + b
AA = EA AND 0b11000
Offset = AA58:60
d[0] = (double) {sign = 0, mantissa =  Offset             , exponent = 0x400}
```

```
d[1] = (double) {sign = 0, mantissa = (Offset+1) AND 0b111, exponent = 0x400}
d[2] = (double) {sign = 0, mantissa = (Offset+2) AND 0b111, exponent = 0x400}
d[3] = (double) {sign = 0, mantissa = (Offset+3) AND 0b111, exponent = 0x400}
```

The following formula is applicable if b is a pointer to a single-precision floating-point value or complex value:

```
EA = a + b
AA = (EA × 2) AND 0b11000
Offset = AA58:60
d[0] = (double) {sign = 0, mantissa =  Offset             , exponent = 0x400}
d[1] = (double) {sign = 0, mantissa = (Offset+1) AND 0b111, exponent = 0x400}
d[2] = (double) {sign = 0, mantissa = (Offset+2) AND 0b111, exponent = 0x400}
d[3] = (double) {sign = 0, mantissa = (Offset+3) AND 0b111, exponent = 0x400}
```

Note:

- AND is the bitwise AND operator.

Example: Loading 8-byte aligned vectors

```
// my_array is an array of the double type
vector4double v, v1, v2, vp;
v1 = vec_ld(0,my_array)    // Load the left part of the vector
v2 = vec_ld(32,my_array)   // Load the right part of the vector
vp = vec_lvsl(0,my_array)  // Generate control value
v  = vec_perm(v1,v2,vp)    // Generate the aligned vector
```

Example: Loading 4-byte aligned vectors

```
// my_array is an array of the float type
vector4double v, v1, v2, vp;
v1 = vec_ld(0,my_array)    // Load the left part of the vector
v2 = vec_ld(16,my_array)   // Load the right part of the vector
vp = vec_lvsl(0,my_array)  // Generate control value
v  = vec_perm(v1,v2,vp)    // Generate the aligned vector
```

vec_lvsr

Purpose

Returns a vector useful for aligning non-aligned data.

Syntax

```
d=vec_lvsr(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	long	double*
		_Complex double*
		float*
		_Complex float*

Result value

The result value is a quad vector. The elements of the quad vector are generated in the following ways:

- Sign: 0
- Mantissa:
  1. For the first element, the mantissa is the result of following operations:
     - If b is a pointer to a double-precision floating-point value or complex value:
       a. Add a and b.
       b. Mask the result of the previous step with 0b11000.
       c. Subtract the result of the previous step from 32.
       d. Take the integer value of bits 58 - 60 from the result of the previous step.
     - If b is a pointer to a single-precision floating-point value or complex value:
       a. Add a and b.
       b. Mask the result of the previous step with 0b1100.
       c. Subtract the result of the previous step from 16.
       d. Take the integer value of bits 59 - 61 from the result of the previous step.
  2. The mantissa is incremented by one for each subsequent element.

     The mantissa is seen as a 3-bit value for the increment operation. That is, incrementing 0b111 produces 0b000.
- Exponent: 0x400

You can use the result as an argument of the vec_perm function.

Formula

The following formula is applicable if b is a pointer to a double-precision floating-point value or complex value:

```
EA = a + b
AA = 32 – (EA AND 0b11000)
Offset = AA58:60
d[0] = (double) {sign = 0, mantissa =  Offset             , exponent = 0x400}
d[1] = (double) {sign = 0, mantissa = (Offset+1) AND 0b111, exponent = 0x400}
d[2] = (double) {sign = 0, mantissa = (Offset+2) AND 0b111, exponent = 0x400}
d[3] = (double) {sign = 0, mantissa = (Offset+3) AND 0b111, exponent = 0x400}
```

The following formula is applicable if b is a pointer to a single-precision floating-point value or complex value:

```
EA = a + b
AA = 16 – (EA AND 0b1100)
Offset = AA59:61
d[0] = (double) {sign = 0, mantissa =  Offset             , exponent = 0x400}
d[1] = (double) {sign = 0, mantissa = (Offset+1) AND 0b111, exponent = 0x400}
d[2] = (double) {sign = 0, mantissa = (Offset+2) AND 0b111, exponent = 0x400}
d[3] = (double) {sign = 0, mantissa = (Offset+3) AND 0b111, exponent = 0x400}
```

Note:

- AND is the bitwise AND operator.

Example: Storing 8-byte aligned vectors

```
void my_vec_store(vector4double v, double *arr)
{
   vector4double v1, v2, v3, p, m1, m2, m3;
   /* generate insert masks */
   p = vec_lvsr(0,arr);
   m1 = vec_cmplt(p,p); /* generate vector of all FALSE */
   m2 = vec_neg(m1);    /* generate vector of all TRUE */
   m3 = vec_perm(m1,m2,p);
   /* get existing data */
   v1 = vec_ld(0,arr);
   v2 = vec_ld(0,arr+4);
   /* permute and insert */
   v3 = vec_perm(v,v,p);
   v1 = vec_sel(v1,v3,m3);
   v2 = vec_sel(v3,v2,m3);
   /* store data back */
   vec_st(0,arr,v1);
   vec_st(0,arr+4,v2);
}
```

Example: Storing 4-byte aligned vectors

```
void my_vec_store(vector4double v, float *arr)
{
   vector4double v1, v2, v3, p, m1, m2, m3
   /* generate insert masks */
   p = vec_lvsr(0,arr);
   m1 = vec_cmplt(p,p); /* generate vector of all FALSE */
   m2 = vec_neg(m1);    /* generate vector of all TRUE */
   m3 = vec_perm(m1,m2,p);
   /* get existing data */
   v1 = vec_ld(0,arr);
   v2 = vec_ld(0,arr+4);
   /* permute and insert */
   v3 = vec_perm(v,v,p);
   v1 = vec_sel(v1,v3,m3);
   v2 = vec_sel(v3,v2,m3);
   /* store data back */
   vec_st(0,arr,v1);
   vec_st(0,arr+4,v2);
}
```

vec_madd

Purpose

Returns a vector containing the results of performing a fused multiply-add operation for each corresponding set of elements of the given vectors.

Syntax

```
d=vec_madd(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	vector4double

Result value

The value of each element of the result is the product of the values of the corresponding elements of a and b, added to the value of the corresponding element of c.

Formula

```
d[0] = ( a[0] × b[0] ) + c[0]
d[1] = ( a[1] × b[1] ) + c[1]
d[2] = ( a[2] × b[2] ) + c[2]
d[3] = ( a[3] × b[3] ) + c[3]
```

Example

```
a =  (10.0, 10.0, 10.0, 10.0)
b =  ( 1.0,  2.0,  3.0,  4.0)
c =  (20.0, 20.0, 20.0, 20.0)
d: (30.0, 40.0, 50.0, 60.0)
```

vec_msub

Purpose

Returns a vector containing the results of performing a multiply-subtract operation using the given vectors.

Syntax

```
d=vec_msub(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	vector4double

Result value

The values of the elements of the result are the product of the values of the corresponding elements of a and b, minus the values of the corresponding elements of c.

Formula

```
d[0] = ( a[0] × b[0] ) - c[0]
d[1] = ( a[1] × b[1] ) - c[1]
d[2] = ( a[2] × b[2] ) - c[2]
d[3] = ( a[3] × b[3] ) - c[3]
```

Example

```
a =  ( 10.0, 10.0, 10.0, 10.0)
b =  (  1.0,  2.0,  3.0,  4.0)
c =  ( 20.0, 20.0, 20.0, 20.0)
d: (-10.0,  0.0, 10.0, 20.0)
```

vec_mul

Purpose

Returns a vector containing the results of performing a multiply operation using the given vectors.

Syntax

```
d=vec_mul(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The values of the elements of the result are obtained by multiplying the elements of a and the corresponding elements of b.

Formula

```
d[0] = a[0] × b[0]
d[1] = a[1] × b[1]
d[2] = a[2] × b[2]
d[3] = a[3] × b[3]
```

Example

```
a =   (10.0,   20.0,   30.0,   40.0)
b =   (50.0,   60.0,   70.0,   80.0)
d: (500.0, 1200.0, 2100.0, 3200.0)
```

vec_nabs

Purpose

Returns a vector containing the results of performing a negative-absolute operation using the given vector.

Syntax

```
d=vec_nabs(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

This function computes the absolute value of each element in the given vector and then assigns the negated value of the result to the corresponding elements in the result vector.

Formula

```
d[0] = -|a[0]|
d[1] = -|a[1]|
d[2] = -|a[2]|
d[3] = -|a[3]|
```

Example

```
a =  ( 10.0, -20.0,  30.0, -40.0)
d: (-10.0, -20.0, -30.0, -40.0)
```

vec_nand

Purpose

Returns a vector containing the results of performing a logical NOT operation of the result of a logical AND operation between the given vectors.

Syntax

```
d=vec_nand(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	**a**	**b**
vector4double	vector4double	vector4double

Result value

The value of each element of the result is the result of a logical NOT operation of a logical AND operation between the corresponding elements of `a` and `b`.

Formula

```
d[0] = NOT (a[0] AND b[0])
d[1] = NOT (a[1] AND b[1])
d[2] = NOT (a[2] AND b[2])
d[3] = NOT (a[3] AND b[3])
```

Example

```
a =  (-1.0, -1.0,  1.0, 1.0)
b =  (-1.0,  1.0, -1.0, 1.0)
d: ( 1.0,  1.0,  1.0,-1.0)
```

vec_not

Purpose

Returns a vector containing the result of a logical NOT operation on the given vector.

Syntax

```
d=vec_not(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

The value of each element of the result is the result of a logical NOT operation of the corresponding element of a.

Formula

```
d[0] = NOT a[0]
d[1] = NOT a[1]
d[2] = NOT a[2]
d[3] = NOT a[3]
```

Example

```
a =  (-1.0, -2.0,  1.0,  2.0)
d: ( 1.0,  1.0, -1.0, -1.0)
```

vec_neg

Purpose

Returns a vector containing the negated value of the corresponding elements in the given vector.

Syntax

```
d=vec_neg(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

This function multiplies the value of each element in the given vector by -1.0 and then assigns the result to the corresponding elements in the result vector.

Formula

```
d[0] = -a[0]
d[1] = -a[1]
d[2] = -a[2]
d[3] = -a[3]
```

Example

```
a =  ( 10.0, -20.0,  30.0, -40.0)
d: (-10.0,  20.0, -30.0,  40.0)
```

vec_nmadd

Purpose

Returns a vector containing the results of performing a negative multiply-add operation on the given vectors.

Syntax

```
d=vec_nmadd(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	vector4double

Result value

The value of each element of the result is the product of the corresponding elements of a and b, added to the corresponding elements of c, and then multiplied by -1.0.

Formula

```
d[0] = - ( ( a[0] × b[0] ) + c[0] )
d[1] = - ( ( a[1] × b[1] ) + c[1] )
d[2] = - ( ( a[2] × b[2] ) + c[2] )
d[3] = - ( ( a[3] × b[3] ) + c[3] )
```

Example

```
a =  ( 10.0,  10.0,  10.0,  10.0)
b =  (  1.0,   2.0,   3.0,   4.0)
c =  ( 20.0,  20.0,  20.0,  20.0)
d: (-30.0, -40.0, -50.0, -60.0)
```

vec_nmsub

Purpose

Returns a vector containing the results of performing a negative multiply-subtract operation on the given vectors.

Syntax

```
d=vec_nmsub(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	vector4double

Result value

The value of each element of the result is the product of the corresponding elements of a and b, subtracted from the corresponding element of c.

Formula

```
d[0] = - ( ( a[0] × b[0] ) - c[0] )
d[1] = - ( ( a[1] × b[1] ) - c[1] )
d[2] = - ( ( a[2] × b[2] ) - c[2] )
d[3] = - ( ( a[3] × b[3] ) - c[3] )
```

Example

```
a =  (10.0, 10.0,  10.0,  10.0)
b =  ( 1.0,  2.0,   3.0,   4.0)
c =  (20.0, 20.0,  20.0,  20.0)
d: (10.0,  0.0, -10.0, -20.0)
```

vec_nor

Purpose

Returns a vector containing the results of performing a logical NOT operation of the result of a logical OR operation between the given vectors.

Syntax

```
d=vec_nor(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is the result of a logical NOT operation of a logical OR operation between the corresponding elements of a and b.

Formula

```
d[0] = NOT (a[0] OR b[0])
d[1] = NOT (a[1] OR b[1])
d[2] = NOT (a[2] OR b[2])
d[3] = NOT (a[3] OR b[3])
```

Example

```
a =  (-1.0, -1.0,  1.0, 1.0)
b =  (-1.0,  1.0, -1.0, 1.0)
d: ( 1.0, -1.0, -1.0,-1.0)
```

vec_or

Purpose

Returns a vector containing the results of performing a logical OR operation between the given vectors.

Syntax

```
d=vec_or(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is the result of a logical OR operation between the corresponding elements of a and b.

Formula

```
d[0] = a[0] OR b[0]
d[1] = a[1] OR b[1]
d[2] = a[2] OR b[2]
d[3] = a[3] OR b[3]
```

Example

```
a =  (-1.0, -1.0,  1.0, 1.0)
b =  (-1.0,  1.0, -1.0, 1.0)
d: (-1.0,  1.0,  1.0, 1.0)
```

vec_orc

Purpose

Returns a vector containing the result of performing a logical OR operation between a and the complement of b.

Syntax

```
d=vec_orc(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is the result of a logical OR operation between the corresponding element of a and the complement of the corresponding element of b.

Formula

```
d[0] = a[0] OR NOT (b[0])
d[1] = a[1] OR NOT (b[1])
d[2] = a[2] OR NOT (b[2])
d[3] = a[3] OR NOT (b[3])
```

Example

```
a =  (-1.0, -1.0,  1.0, 1.0)
b =  (-1.0,  1.0, -1.0, 1.0)
d: ( 1.0, -1.0,  1.0, 1.0)
```

vec_perm

Purpose

Returns a vector that contains some elements of two vectors, in the order specified by a third vector.

Syntax

```
d=vec_perm(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	vector4double

Result value

The value of each element of the result is the element of the concatenation of a and b that is specified by bits 0:2 of the mantissa of the corresponding element of c.

Each element of c must have an exponent equal to 0x400, or the corresponding element of the result is undefined.

Note: The following functions generate control values that can be used for c:

- "vec_gpci" on page 433
- "vec_lvsl" on page 441
- "vec_lvsr" on page 443

Formula

```
Concat = ( a[0], a[1], a[2], a[3],
           b[0], b[1], b[2], b[3] )
d[0] = Concat[Mantissa02(c[0])]
d[1] = Concat[Mantissa02(c[1])]
d[2] = Concat[Mantissa02(c[2])]
d[3] = Concat[Mantissa02(c[3])]
```

Note:

Mantissa02 is a function that returns the integer that is equivalent to the bits 0:2 of the mantissa of its argument.

Example

If a = (10.0, 20.0, 30.0, 40.0), b = (50.0, 60.0, 70.0, 80.0), and the mantissas of the elements of c = (2,3,4,5), the result value is (30.0, 40.0, 50.0, 60.0).

vec_promote

Purpose

Returns a vector with a in element position b.

Syntax

```
d=vec_promote(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	double	int

Result value

The result is a vector with a in element position b. This function uses modulo arithmetic on b to determine the element number. For example, if b is out of range, the compiler uses b modulo the number of elements in the vector to determine the element position. The other elements of the vector are undefined.

Formula

```
d[b MOD 4] = a
```

Note: `MOD` is the modulo operator.

Example

```
a = 50.0
b = 1
d: ( X, 50.0, Y, Z) // X, Y, and Z are undefined values
```

vec_re

Purpose

Returns a vector containing estimates of the reciprocals of the corresponding elements of the given vector.

Syntax

```
d=vec_re(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of the result contains the estimated value of the reciprocal of the corresponding element of `a`.

Note:

The precision guarantee is specified by the following expression, where x is the value of each element of `a` and r is the value of the corresponding element of the result value:

$| (r\text{-}1/x) / (1/x) | \leq 1/256$

Special operands

Special operands are handled as follows:

Operand	Estimate	Exception
-Infinity	-0	None
-0	-Infinity[1]	ZX
+0	+Infinity[1]	ZX
+Infinity	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

Operand	Estimate	Exception
1. No result if $FPSCR_{ZE}$ = 1. 2. No result if $FPSCR_{VE}$ = 1.		

Formula

```
d[0] = 1 / a[0]
d[1] = 1 / a[1]
d[2] = 1 / a[2]
d[3] = 1 / a[3]
```

Example

```
a =  (2.0,  4.0, 5.0,   8.0)
d: (0.5, 0.25, 0.2, 0.125)
```

vec_res

Purpose

Returns a vector containing estimates of the reciprocals of the corresponding elements of the given vector.

Syntax

```
d=vec_res(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

The double-precision elements of `a` are first truncated to single-precision values. An estimate of the reciprocal of each single-precision element of `a` is then converted to double precision and saved in the corresponding element of the result.

Note:

The precision guarantee is specified by the following expression, where x is the value of each element of `a` and r is the value of the corresponding element of the result value:

$| (r\text{-}1/x) / (1/x) | \leq 1/256$

Special operands

Special operands are handled as follows:

Operand	Estimate	Exception
-Infinity	-0	None

Operand	Estimate	Exception
-0	-Infinity[1]	ZX
+0	+Infinity[1]	ZX
+Infinity	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if $FPSCR_{ZE}$ = 1.
2. No result if $FPSCR_{VE}$ = 1.

Formula

```
d[0] = (double) (1 / (float) a[0])
d[1] = (double) (1 / (float) a[1])
d[2] = (double) (1 / (float) a[2])
d[3] = (double) (1 / (float) a[3])
```

Example

```
a =  (2.0,  4.0, 5.0,   8.0)
d: (0.5, 0.25, 0.2, 0.125)
```

vec_round

Purpose

Returns a vector containing the rounded values of the corresponding elements of the given vector.

Syntax

```
d=vec_round(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of the result contains the value of the corresponding element of a, rounded to the nearest representable floating-point integer.

Formula

For each element of a:

```
If a[n] <0, d[n] = (a[n] – 0.5), truncated to the nearest integral value.
If a[n] >0, d[n] = (a[n] + 0.5), truncated to the nearest integral value.
If a[n] EQ 0, d[n] = 0.
```

Note: EQ is the equal operator.

Example

```
ARG1  = (-5.8, -2.3, 2.3, 5.8)
Result: (-6.0, -2.0, 2.0, 6.0)
```

vec_rsp

Purpose

Returns a vector containing the single-precision values of the corresponding elements of the given vector.

Syntax

```
d=vec_rsp(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

The value of each element of the result contains the single-precision value of the corresponding element of a.

Formula

```
d[0] = (double) ( (float) a[0] )
d[1] = (double) ( (float) a[1] )
d[2] = (double) ( (float) a[2] )
d[3] = (double) ( (float) a[3] )
```

vec_rsqrte

Purpose

Returns a vector containing estimates of the reciprocal square roots of the corresponding elements of the given vector.

Syntax

```
d=vec_rsqrte(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of the result contains the estimated value of the reciprocal square root of the corresponding element of `a`.

Note:

The precision guarantee is specified by the following expression, where x is the value of each element of `a` and r is the value of the corresponding element of the result value:

$| (r\text{-}1/\sqrt{x}) / 1/\sqrt{x} | \leq 1/32$

Special operands

Special operands are handled as follows:

Operand	Estimate	Exception
-Infinity	QNaN[2]	VXSQRT
<0	QNaN[2]	VXSQRT
-0	-Infinity[1]	ZX
+0	+Infinity[1]	ZX
+Infinity	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if $FPSCR_{ZE} = 1$.
2. No result if $FPSCR_{VE} = 1$.

Formula

```
d[0] = 1 / √a[0]
d[1] = 1 / √a[1]
d[2] = 1 / √a[2]
d[3] = 1 / √a[3]
```

Example

```
a =  (4.0, 16.0, 25.0,  64.0)
d: (0.5, 0.25,  0.2, 0.125)
```

vec_rsqrtes

Purpose

Returns a vector containing estimates of the reciprocal square roots of the corresponding elements of the given vector.

Syntax

```
d=vec_rsqrtes(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

The double-precision elements of a are first truncated to single-precision values. An estimate of the reciprocal square root of each single-precision element of a is then converted to double precision and saved in the corresponding element of the result.

Note:

The precision guarantee is specified by the following expression, where x is the value of each element of a and r is the value of the corresponding element of the result value:

$| (r\text{-}1/\sqrt{x}) \ / \ 1/\sqrt{x} | \leq 1/32$

Special operands

Special operands are handled as follows:

Operand	Estimate	Exception
-Infinity	QNaN[2]	VXSQRT
<0	QNaN[2]	VXSQRT
-0	-Infinity[1]	ZX
+0	+Infinity[1]	ZX
+Infinity	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if $FPSCR_{ZE} = 1$.
2. No result if $FPSCR_{VE} = 1$.

Formula

```
d[0] = (double) (1 / √ (float) a[0])
d[1] = (double) (1 / √ (float) a[1])
d[2] = (double) (1 / √ (float) a[2])
d[3] = (double) (1 / √ (float) a[3])
```

Example

```
a =  (4.0, 16.0, 25.0,  64.0)
d: (0.5, 0.25,  0.2, 0.125)
```

vec_sel

Purpose

Returns a vector containing the value of either a or b depending on the value of c.

Syntax

```
d=vec_sel(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	vector4double

Result value

The value of each element of the result is equal to the corresponding element of b if the corresponding element of c is greater than or equal to zero (regardless of sign), or the value is equal to the corresponding element of a if the corresponding element of c is less than zero or NaN.

Formula

```
If (c[0] ≥ 0) Then d[0] = b[0] Else d[0] = a[0]
If (c[1] ≥ 0) Then d[1] = b[1] Else d[1] = a[1]
If (c[2] ≥ 0) Then d[2] = b[2] Else d[2] = a[2]
If (c[3] ≥ 0) Then d[3] = b[3] Else d[3] = a[3]
```

Example

```
a =  (20.0,  20.0, 20.0,  20.0)
b =  (10.0,  10.0, 10.0,  10.0)
c =  ( 1.0,  -1.0,  2.5,  -2.5)
d: (10.0,  20.0, 10.0,  20.0)
```

vec_sldw

Purpose

Returns a vector by concatenating a and b, and then left-shifting the result vector by multiples of 8 bytes. c specifies the offset for the shifting operation.

Syntax

```
d=vec_sldw(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	int, a value in 0 - 3

Result value

After left-shifting the concatenated a and b by multiples of 8 bytes specified by c, the function takes the four leftmost 8-byte values and forms the result vector.

Formula

```
Concat = ( a[0], a[1], a[2], a[3],
           b[0], b[1], b[2], b[3] )
d[0] = Concat[c]
d[1] = Concat[c+1]
d[2] = Concat[c+2]
d[3] = Concat[c+3]
```

Example

```
a = (10.0, 20.0, 30.0, 40.0)
b = (50.0, 60.0, 70.0, 80.0)
c = 2
d: (30.0, 40.0, 50.0, 60.0)
```

vec_splat

Purpose

Returns a vector that has all of its elements set to a given value.

Syntax

```
d=vec_splat(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	**a**	**b**
vector4double	vector4double	int, a value in 0 - 3

Result value

The value of each element of the result is the value of the element of a specified by b.

Formula

```
d[0] = a[b]
d[1] = a[b]
d[2] = a[b]
d[3] = a[b]
```

Example

```
a = (10.0, 20.0, 30.0, 40.0)
b = 1
d: (20.0, 20.0, 20.0, 20.0)
```

vec_splats

Purpose

Returns a vector of which the value of each element is set to a.

Syntax

```
d=vec_splats(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	double

Result value

The value of each element of the result is a.

Formula

```
d[0] = a
d[1] = a
d[2] = a
d[3] = a
```

Example

```
a = 50.0
d: (50.0, 50.0, 50.0, 50.0)
```

vec_st, vec_sta

Purpose

Stores a vector to memory at the given address.

Syntax

```
vec_st(a, b, c)
vec_sta(a, b, c)
```

Argument types

The following table describes the types of the function arguments.

a	b	c
vector4double	long	int*
		unsigned*
		long*
		unsigned long*
		long long*
		unsigned long long*
		float*
		_Complex float*
		double*
		_Complex double*

Result

The effective address (EA) is the sum of `b` and `c`. The effective address is truncated to an *n*-byte alignment depending on the type of `c` as shown in the following table. The value of `a` is then stored at the effective address.

Type of c	*n*
`int*` `unsigned*`	16

`long*`
`unsigned long*`
`long long*`
`unsigned long long*`
	32

`float*`	16
`_Complex float*`	
`double*`	32
`_Complex double*`	

`vec_sta` generates an exception (SIGBUS) if the effective address is not aligned to the appropriate memory boundary indicated in the table.

If `c` is a pointer to a variable of single-precision floating-point type or single-precision complex type, the elements of `a` are converted to single precision before being saved to memory.

If `c` is a pointer to a variable of 4-byte integer type, the four low-order bytes of the elements of `a` are saved to memory.

Formula

The following table shows the formulas depending on the type of `c`.

Type of c	**Formula**
`int*` `unsigned*`	$Memory\_4B[EA]=a[0]_{32:63}$ $Memory\_4B[EA+4]=a[1]_{32:63}$ $Memory\_4B[EA+8]=a[2]_{32:63}$ $Memory\_4B[EA+12]=a[3]_{32:63}$

`long*`
`unsigned long*`
`long long*`
`unsigned long long*`
	`Memory[EA]=a[0]`
	`Memory[EA+8]=a[1]`
	`Memory[EA+16]=a[2]`
	`Memory[EA+24]=a[3]`

`float*`	`Memory_SP[EA]=(float) a[0]`
	`Memory_SP[EA+4]=(float) a[1]`
	`Memory_SP[EA+8]=(float) a[2]`
	`Memory_SP[EA+12]=(float) a[3]`

`_Complex float*`	

`double*`	`Memory[EA]=a[0]`
	`Memory[EA+8]=a[1]`
	`Memory[EA+16]=a[2]`
	`Memory[EA+24]=a[3]`

`_Complex double*`	

Notes:

- `Memory_SP[]` is a single-precision floating-point array.
- `Memory_4B[]` is a 4-byte integer array.

Examples

Type of c	a	Memory values
int* unsigned*	(10, 20, 30, 40)	10, 20, 30, 40
long* unsigned long* long long* unsigned long long*	(10.0, 20.0, 30.0, 40.0)	0x4024000000000000, 0x4034000000000000, 0x403E000000000000, 0x4044000000000000
float*	(10.0, 20.0, 30.0, 40.0)	10.0f, 20.0f, 30.0f, 40.0f
_Complex float*		(10.0f, 20.0f) (30.0f, 40.0f)
double*		10.0, 20.0, 30.0, 40.0
_Complex double*		(10.0, 20.0) (30.0, 40.0)

vec_st2, vec_st2a

Purpose

Stores the first two elements of a quad vector to memory at the given address.

Syntax

```
vec_st2(a, b, c)
vec_st2a(a, b, c)
```

Argument types

The following table describes the types of the function arguments.

a	b	c
vector4double	long	double*
		float*

Result

The effective address (EA) is the sum of b and c. The effective address is truncated to an *n*-byte alignment depending on the type of c as shown in the following table. The first two elements of a are then stored at the effective address.

	Type of c	
	double*	**float***
n	16	8

vec_st2a generates an exception (SIGBUS) if the effective address is not aligned to the appropriate memory boundary indicated in the table.

If c is a pointer to a variable of single-precision floating-point type, the elements of a are converted to single precision before being saved to memory.

Formula

The following table shows the formulas depending on the type of c.

	Type of c	
	double*	**float***
Formula	Memory[EA]=a[0] Memory[EA+8]=a[1]	Memory_SP[EA]=(float) a[0] Memory_SP[EA+4]=(float) a[1]

Note: Memory_SP[] is a single-precision floating-point array.

Examples

	Type of c	
	double*	**float***
a	(10.0, 20.0, 30.0, 40.0)	
Memory values	10.0, 20.0	10.0f, 20.0f

vec_sts, vec_stsa

Purpose

Stores the first element or the first two elements of a quad vector to memory at the given address.

Syntax

```
vec_sts(a, b, c)
vec_stsa(a, b, c)
```

Argument types

The following table describes the types of the function arguments.

a	b	c
vector4double	long	double* (only for vec_sts)
		float* (only for vec_sts)
		_Complex double*
		_Complex float*

Result

The effective address (EA) is the sum of b and c. If c is a pointer to a complex value, the effective address is truncated to an *n*-byte alignment depending on the type of c as shown in the following table. The value of a is then stored to the effective address as follows:

- If c is a pointer to a variable of floating-point type, the first element of a is stored to memory.
- If c is a pointer to a variable of complex type, the first two elements of a are stored to memory.

	Type of c	
	_Complex double*	_Complex float*
n	16	8

vec_stsa generates an exception (SIGBUS) if the effective address is not aligned to the appropriate memory boundary indicated in the table.

If c is a pointer to a variable of single-precision floating-point type or single-precision complex type, the elements of a are converted to single precision before being saved to memory.

Formula

The following tables show the formulas depending on the type of c.

	Type of c			
	double*	_Complex double*	float*	_Complex float*
Formula	Memory[EA] = a[0]	Memory[EA] = a[0] Memory[EA+8] = a[1]	Memory_SP[EA] = (float) a[0]	Memory_SP[EA] = (float) a[0] Memory_SP[EA+4] = (float) a[1]

Note: Memory_SP[] is a single-precision floating-point array.

Examples

	Type of c			
	double*	_Complex double*	float*	_Complex float*
a	(10.0, 20.0, 30.0, 40.0)			
Memory values	10.0	(10.0, 20.0)	10.0f	(10.0f, 20.0f)

vec_sub

Purpose

Returns a vector containing the result of subtracting each element of b from the corresponding element of a.

Syntax

```
d=vec_sub(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is the result of subtracting the value of the corresponding element of b from the value of the corresponding element of a.

Formula

```
d[0] = a[0] - b[0]
d[1] = a[1] - b[1]
d[2] = a[2] - b[2]
d[3] = a[3] - b[3]
```

Example

```
a =  (50.0, 60.0, 70.0, 80.0)
b =  (10.0, 20.0, 30.0, 40.0)
d: (40.0, 40.0, 40.0, 40.0)
```

vec_swdiv, vec_swdiv_nochk

Purpose

Returns a vector containing the result of dividing each element of a by the corresponding element of b.

Syntax

```
d=vec_swdiv(a, b)
d=vec_swdiv_nochk(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

For vec_swdiv_nochk, the compiler does not check the validity of the arguments. You must ensure that the following conditions are satisfied where x represents each element of a and y represents the corresponding element of b:

- $2^{-1021} \le |y| \le 2^{1020}$
- If $x \ne 0.0$

  $2^{-969} \le |x| <$ Infinity

  $2^{-1020} \le |x / y| \le 2^{1022}$

Result value

The values of the elements of the result are obtained by dividing the elements of a by the corresponding elements of b.

When the following options are used, the result is bitwise identical to the IEEE division.

- **-qstrict=precision**
- **-qstrict=ieeefp**
- **-qstrict=zerosigns**

- **-qstrict=operationprecision**

Otherwise, the result might differ slightly from the IEEE division.

Formula

```
d[0] = a[0] / b[0]
d[1] = a[1] / b[1]
d[2] = a[2] / b[2]
d[3] = a[3] / b[3]
```

Example

```
a =  (50.0,  1.0,  30.0,  40.0)
b =  (10.0,  5.0,  -1.0,  80.0)
d: ( 5.0,  0.2, -30.0,   0.5)
```

vec_swdivs, vec_swdivs_nochk

Purpose

Returns a vector containing the result of dividing each element of `a` by the corresponding element of `b`.

Syntax

```
d=vec_swdivs(a, b)
d=vec_swdivs_nochk(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

For `vec_swdivs_nochk`, the compiler does not check the validity of the arguments. You must ensure that the following conditions are satisfied where x represents each element of `a` and y represents the corresponding element of `b`:

- $2^{-125} \le |y| \le 2^{124}$
- If $x \ne 0$

  $2^{-102} \le |x| <$ Infinity

  $2^{-124} \le |x / y| \le 2^{126}$

Result value

The double-precision elements of `a` and `b` are first truncated to single-precision values. The result of dividing the single-precision elements of `a` by the corresponding single-precision elements of `b` is then converted to double precision and saved in the corresponding elements of the result.

When the following options are used, the result is bitwise identical to the IEEE division.

- **-qstrict=precision**
- **-qstrict=ieeefp**

- **-qstrict=zerosigns**
- **-qstrict=operationprecision**

Otherwise, the result might differ slightly from the IEEE division.

Formula

```
d[0] = (double) ( (float) a[0] / (float) b[0] )
d[1] = (double) ( (float) a[1] / (float) b[1] )
d[2] = (double) ( (float) a[2] / (float) b[2] )
d[3] = (double) ( (float) a[3] / (float) b[3] )
```

Example

```
a =  (50.0,  1.0,  30.0,  40.0)
b =  (10.0,  5.0,  -1.0,  80.0)
d: ( 5.0,  0.2, -30.0,   0.5)
```

vec_swsqrt, vec_swsqrt_nochk

Purpose

Returns a vector containing the square root of each element in the given vector.

Syntax

```
d=vec_swsqrt(a)
d=vec_swsqrt_nochk(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	**a**
vector4double	vector4double

For `vec_swsqrt_nochk`, the compiler does not check the validity of the arguments. You must ensure that the following condition is satisfied where *x* represents each element of `a`:

- 2^{-969} <= *x* < Infinity

Result value

The result value is a quad vector that contains the square root of each element of `a`.

When the following options are used, the result is bitwise identical to the IEEE square root.

- **-qstrict=precision**
- **-qstrict=ieeefp**
- **-qstrict=zerosigns**
- **-qstrict=operationprecision**

Otherwise, the result might differ slightly from the IEEE square root.

Formula

```
d[0] = √a[0]
d[1] = √a[1]
d[2] = √a[2]
d[3] = √a[3]
```

Example

```
a =  ( 4.0,  9.0,  16.0,  25.0)
d: ( 2.0,  3.0,   4.0,   5.0)
```

vec_swsqrts, vec_swsqrts_nochk

Purpose

Returns a vector containing estimates of the square roots of the corresponding elements of the given vector.

Syntax

```
d=vec_swsqrts(a)
d=vec_swsqrts_nochk(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

For vec_swsqrts_nochk, the compiler does not check the validity of the arguments. You must ensure that the following condition is satisfied where *x* represents each element of a:

- $2^{-102} <= x <$ Infinity

Result value

The double-precision elements of a are first truncated to single-precision values. The square root of each single-precision element of a is then converted to double-precision and saved in the corresponding element of the result.

When the following options are used, the result is bitwise identical to the IEEE square root.

- **-qstrict=precision**
- **-qstrict=ieeefp**
- **-qstrict=zerosigns**
- **-qstrict=operationprecision**

Otherwise, the result might differ slightly from the IEEE square root.

Formula

```
d[0] = (double) √ ((float) a[0])
d[1] = (double) √ ((float) a[1])
d[2] = (double) √ ((float) a[2])
d[3] = (double) √ ((float) a[3])
```

Example

```
a =  ( 4.0,  9.0,  16.0,  25.0)
d: ( 2.0,  3.0,   4.0,   5.0)
```

vec_trunc

Purpose

Returns a vector containing the truncated values of the corresponding elements of the given vector.

Syntax

```
d=vec_trunc(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector4double	vector4double

Result value

Each element of the result contains the value of the corresponding element of a, truncated to an integral value.

Example

```
a  = (-5.8, -2.3, 2.3, 5.8)
d: (-5.0, -2.0, 2.0, 5.0)
```

vec_tstnan

Purpose

Returns a vector whose elements depend on if the value of the corresponding element of a or b is NaN.

Syntax

```
d=vec_tstnan(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is 1.0 if the corresponding element of a or b is a NaN, otherwise the value is -1.0.

Formula

```
If ((a[0] EQ NaN) or (b[0] EQ NaN)) Then d[0] = 1.0 Else d[0] = -1.0
If ((a[1] EQ NaN) or (b[1] EQ NaN)) Then d[1] = 1.0 Else d[1] = -1.0
If ((a[2] EQ NaN) or (b[2] EQ NaN)) Then d[2] = 1.0 Else d[2] = -1.0
If ((a[3] EQ NaN) or (b[3] EQ NaN)) Then d[3] = 1.0 Else d[3] = -1.0
```

Note: EQ is the equal operator.

Example

```
a =  (10.0, 20.0,  NaN, 40.0)
b =  (50.0,  NaN, 70.0, 80.0)
d: (-1.0,  1.0,  1.0, -1.0)
```

vec_xmadd

Purpose

Returns a vector containing the results of performing a fused cross multiply-add operation for each corresponding set of elements of the given vectors.

Syntax

```
d=vec_xmadd(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	vector4double

Result value

The values of the elements of the result are the product of the values of the first and the third elements of a and the elements of b, added to the values of the corresponding elements of c.

Formula

```
d[0] = ( a[0] × b[0] ) + c[0]
d[1] = ( a[0] × b[1] ) + c[1]
d[2] = ( a[2] × b[2] ) + c[2]
d[3] = ( a[2] × b[3] ) + c[3]
```

Example

```
a =  ( 1.0,  0.0,  3.0,  0.0)
b =  ( 5.0, 10.0, 15.0, 20.0)
c =  (10.0, 10.0, 10.0, 10.0)
d: (15.0, 20.0, 55.0, 70.0)
```

vec_xmul

Purpose

Returns a vector containing the result of cross multiplying the first and the third elements of a by the elements of b.

Syntax

```
d=vec_xmul(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The values of the elements of the result are obtained by cross multiplying the first and the third elements of a by the elements of b.

Formula

```
d[0] = a[0] × b[0]
d[1] = a[0] × b[1]
d[2] = a[2] × b[2]
d[3] = a[2] × b[3]
```

Example

```
a =   (10.0,   0.0,   30.0,    0.0)
b =   (50.0,  60.0,   70.0,   80.0)
d: (500.0, 600.0, 2100.0, 2400.0)
```

vec_xor

Purpose

Returns a vector containing the results of performing a logical exclusive OR operation between the given vectors.

Syntax

```
d=vec_xor(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector4double	vector4double	vector4double

Result value

The value of each element of the result is the result of a logical exclusive OR between the corresponding elements of a and b.

Formula

```
d[0] = a[0] XOR b[0]
d[1] = a[1] XOR b[1]
d[2] = a[2] XOR b[2]
d[3] = a[3] XOR b[3]
```

Example

```
a =  (-1.0, -1.0,  1.0,  1.0)
b =  (-1.0,  1.0, -1.0,  1.0)
d: (-1.0,  1.0,  1.0, -1.0)
```

vec_xxcpnmadd

Purpose

Returns a vector containing the results of performing a fused double cross conjugate multiply/add for each corresponding set of elements of the given vectors.

Syntax

```
d=vec_xxcpnmadd(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	vector4double

Result value

The values of the elements of the result are specified in the formula.

Formula

```
d[0] =   ( ( a[1] × b[1] ) + c[0] )
d[1] = - ( ( a[0] × b[1] ) + c[1] )
d[2] =   ( ( a[3] × b[3] ) + c[2] )
d[3] = - ( ( a[2] × b[3] ) + c[3] )
```

Example

```
a =  (  1.0,   2.0,  3.0,   4.0)
b =  (  0.0,  10.0,  0.0,  20.0)
c =  ( 10.0,  10.0, 10.0,  10.0)
d: ( 30.0, -20.0, 90.0, -70.0)
```

vec_xxmadd

Purpose

Returns a vector containing the results of performing a fused double cross multiply-add operation for each corresponding set of elements of the given vectors.

Syntax

```
d=vec_xxmadd(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	vector4double

Result value

The values of the elements of the result are specified in the formula.

Formula

```
d[0] = ( a[1] × b[1] ) + c[0]
d[1] = ( a[0] × b[1] ) + c[1]
d[2] = ( a[3] × b[3] ) + c[2]
d[3] = ( a[2] × b[3] ) + c[3]
```

Example

```
a =  (  1.0,  2.0,  3.0,  4.0)
b =  (  0.0, 10.0,  0.0, 20.0)
c =  ( 10.0, 10.0, 10.0, 10.0)
d: ( 30.0, 20.0, 90.0, 70.0)
```

vec_xxnpmadd

Purpose

Returns a vector containing the results of performing a fused double cross complex multiply-add operation for each corresponding set of elements of the given vectors.

Syntax

```
d=vec_xxnpmadd(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector4double	vector4double	vector4double	vector4double

Result value

The values of the elements of the result are specified in the formula.

Formula

```
d[0] = - ( ( a[1] × b[1] ) + c[0] )
d[1] =   ( ( a[0] × b[1] ) + c[1] )
d[2] = - ( ( a[3] × b[3] ) + c[2] )
d[3] =   ( ( a[2] × b[3] ) + c[3] )
```

Example

```
a =  (   1.0,  2.0,   3.0,  4.0)
b =  (   0.0, 10.0,   0.0, 20.0)
c =  (  10.0, 10.0,  10.0, 10.0)
d: ( -30.0, 20.0, -90.0, 70.0)
```

GCC atomic memory access built-in functions

This section provides reference information for atomic memory access built-in functions whose behavior corresponds to that provided by GNU Compiler Collection (GCC). In a program with multiple threads, you can use these functions to atomically and safely modify data in one thread without interference from other threads.

These built-in functions manipulate data atomically, regardless of how many processors are installed in the host machine.

In the prototype of each function, the parameter types *T*, *U*, and *V* can be of pointer or integral type. *U* and *V* can also be of real floating-point type, but only when *T* is of integral type. The following tables list the integral and floating-point types that are supported by these built-in functions.

Table 46. Supported integral data types

`signed char`	`unsigned char`
`short int`	`unsigned short int`
`int`	`unsigned int`
`long int`	`unsigned long int`
`long long int`	`unsigned long long int`
C++ `bool`	C `_Bool`

Table 47. Supported floating-point data types

`float`
`double`
`long double`

In the prototype of each function, the ellipsis (...) represents an optional list of parameters. XL C/C++ ignores these optional parameters and protects all globally accessible variables.

The GCC atomic memory access built-in functions are grouped into the following categories.

Atomic lock, release, and synchronize functions

__sync_lock_test_and_set
Purpose

This function atomically assigns the value of *__v* to the variable that *__p* points to.

An acquire memory barrier is created when this function is invoked.

Prototype

T __sync_lock_test_and_set (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of the variable that is to be set.

__v
The value to set to the variable that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_lock_release
Purpose

This function releases the lock acquired by the `__sync_lock_test_and_set` function, and assigns the value of zero to the variable that *__p* points to.

A release memory barrier is created when this function is invoked.

Prototype

void __sync_lock_release (*T** *__p*, ...);

Parameters

__p
The pointer of the variable that is to be set.

__sync_synchronize
Purpose

This function synchronizes data in all threads.

A full memory barrier is created when this function is invoked.

Prototype

void __sync_synchronize ();

Atomic fetch and operation functions

__sync_fetch_and_and
Purpose

This function performs an atomic bitwise AND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_and (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable on which the bitwise AND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
The variable with which the bitwise AND operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_nand
Purpose

This function performs an atomic bitwise NAND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_nand (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable on which the bitwise NAND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
The variable with which the bitwise NAND operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_or
Purpose

This function performs an atomic bitwise inclusive OR operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_or (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable on which the bitwise inclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
The variable with which the bitwise inclusive OR operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_xor
Purpose

This function performs an atomic bitwise exclusive OR operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_xor (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable on which the bitwise exclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
The variable with which the bitwise exclusive OR operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_add
Purpose

This function atomically adds the value of *__v* to the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_add (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable to which *__v* is to be added. The value of this variable is to be changed to the result of the add operation.

__v
The variable whose value is to be added to the variable that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_sub
Purpose

This function atomically subtracts the value of *__v* from the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_sub (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable from which *__v* is to be subtracted. The value of this variable is to be changed to the result of the sub operation.

__v
The variable whose value is to be subtracted from the variable that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

Atomic operation and fetch functions

__sync_and_and_fetch
Purpose

This function performs an atomic bitwise AND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_and_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable on which the bitwise AND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
The variable with which the bitwise AND operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_nand_and_fetch
Purpose

This function performs an atomic bitwise NAND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_nand_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable on which the bitwise NAND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
The variable with which the bitwise NAND operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_or_and_fetch
Purpose

This function performs an atomic bitwise inclusive OR operation on the variable *__v* with variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_or_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable on which the bitwise inclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
The variable with which the bitwise inclusive OR operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_xor_and_fetch
Purpose

This function performs an atomic bitwise exclusive OR operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_xor_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of the variable on which the bitwise exclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
The variable with which the bitwise exclusive OR operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_add_and_fetch
Purpose

This function atomically adds the value of *__v* to the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_add_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable to which *__v* is to be added. The value of this variable is to be changed to the result of the add operation.

__v
The variable whose value is to be added to the variable that *__p* points to.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_sub_and_fetch
Purpose

This function atomically subtracts the value of *__v* from the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_sub_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of a variable from which *__v* is to be subtracted. The value of this variable is to be changed to the result of the sub operation.

__v
The variable whose value is to be subtracted from the variable that *__p* points to.

Return value

The function returns the new value of the variable that *__p* points to.

Atomic compare and swap functions

__sync_val_compare_and_swap
Purpose

This function compares the value of *__compVal* to the value of the variable that *__p* points to. If they are equal, the value of *__exchVal* is stored in the address that is specified by *__p*; otherwise, no operation is performed.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_val_compare_and_swap (*T** *__p*, *U* *__compVal*, *V* *__exchVal*, ...);

Parameters

__p
The pointer to a variable whose value is to be compared with.

__compVal
The value to be compared with the value of the variable that *__p* points to.

__exchVal
The value to be stored in the address that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_bool_compare_and_swap
Purpose

This function compares the value of *__compVal* with the value of the variable that *__p* points to. If they are equal, the value of *__exchVal* is stored in the address that is specified by *__p*; otherwise, no operation is performed.

A full memory barrier is created when this function is invoked.

Prototype

bool __sync_bool_compare_and_swap (*T** *__p*, *U* *__compVal*, *V* *__exchVal*, ...);

Parameters

__p
The pointer to a variable whose value is to be compared with.

__compVal
The value to be compared with the value of the variable that *__p* points to.

__exchVal
The value to be stored in the address that *__p* points to.

Return value

If the value of *__compVal* and the value of the variable that *__p* points to are equal, the function returns `true`; otherwise, it returns `false`.

Miscellaneous built-in functions

Miscellaneous functions are grouped into the following categories:

- "Optimization-related functions"
- "Move to/from register functions" on page 486
- "Memory-related functions" on page 488

Optimization-related functions

__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
Must be a constant literal.

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
Purpose

Move from Time Base

In 64-bit mode, returns the entire doubleword of the time base register.

Prototype

```
unsigned long __mftb (void);
```

Usage

It is recommended that you insert the `__fence` built-in function before and after the `__mftb` built-in function.

__mftbu
Purpose

Move from Time Base Upper

Returns the upper word of the time base register.

Prototype

unsigned int __mftbu (void);

Usage

It is recommended that you insert the `__fence` built-in function before and after the `__mftbu` built-in function.

__mtdcr
Purpose

Move to Device Control Register

Sets the value of the device control register *registerNumber* with unsigned long *value*. The *registerNumber* must be known at compile time. Valid only for PowerPC® 440.

Prototype

void __mtdcr(const int *registerNumber*, unsigned long *value*);

Parameters

registerNumber
The *registerNumber* must be known at compile time.

value
Must be known at compile time.

__mfmsr
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
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
Purpose

Move to Machine State Register

Moves the contents of bits 32 to 62 of the designated GPR into the MSR.

Prototype

```
void __mtmsr (unsigned long value);
```

Parameters

value
The bitwise OR result of bits 48 and 49 of *value* is placed into MSR_{48}. The bitwise OR result of bits 58 and 49 of *value* is placed into MSR_{58}. The bitwise OR result of bits 59 and 49 of *value* is placed into MSR_{59}. Bits 32:47, 49:50, 52:57, and 60:62 of *value* are placed into the corresponding bits of the MSR.

Usage

Execution of this instruction is privileged and restricted to supervisor mode only.

__mtspr
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.

__mem_delay
Purpose

The **__mem_delay** built-in function specifies how many delay cycles there are for specific loads. These specific loads are delinquent loads with a long memory access latency because of cache misses.

When you specify which load is delinquent the compiler takes that information and carries out optimizations such as data prefetching.

Prototype

void* __mem_delay (const void *address, const unsigned int *cycles*);

Parameters

address
The address of the data to be loaded or stored.

cycles
A compile time constant, typically either L1 miss latency or L2 miss latency.

Usage

The **__mem_delay** built-in function is placed immediately before a statement that contains a specified memory reference.

Examples

Here is how you generate code using assist threads with **__mem_delay**:

Initial code:

```
int y[64], x[1089], w[1024];

  void foo(void){
    int i, j;
    for (i = 0; i &l; 64; i++) {
      for (j = 0; j < 1024; j++) {

        /* what to prefetch? y[i]; inserted by the user */
        __mem_delay(&y[i], 10);
        y[i] = y[i] + x[i + j] * w[j];
        x[i + j + 1] = y[i] * 2;
    }
  }
}
```

Assist thread generated code:

```
void foo@clone(unsigned thread_id, unsigned version)

{ if (!1) goto lab_1;

/* version control to synchronize assist and main thread */
if (version == @2version0) goto lab_5;

goto lab_1;

lab_5:

@CIV1 = 0;

do { /* id=1 guarded */ /* ~2 */

if (!1) goto lab_3;

@CIV0 = 0;

do { /* id=2 guarded */ /* ~4 */

/* region = 0 */

/* __dcbt call generated to prefetch y[i] access */
__dcbt(((char *)&y + (4)*(@CIV1)))
@CIV0 = @CIV0 + 1;
} while ((unsigned) @CIV0 < 1024u); /* ~4 */

lab_3:
@CIV1 = @CIV1 + 1;
} while ((unsigned) @CIV1 < 64u); /* ~2 */

lab_1:

return;
}
```

Built-in functions for parallel processing

Use these built-in functions to obtain information about the parallel environment:

• "OpenMP runtime functions" on page 499

IBM SMP built-in functions

__parthds (C only)

Purpose

Returns the value of the **parthds** runtime option.

Prototype

```
int __parthds (void);
```

Return value

If the **parthds** option is not explicitly set, returns the default value set by the runtime library. If the **-qsmp** compiler option was not specified during program compilation, returns 1 regardless of runtime options selected.

__usrthds (C only)

Purpose

Returns the value of the **usrthds** runtime option.

Prototype

```
int __usrthds (void);
```

Return value

If the **usrthds** option is not explicitly set, or the **-qsmp** compiler option is not specified during program compilation, returns `0` regardless of runtime options selected.

Built-in functions for thread-level speculative execution

To call the built-in functions that are specific to thread-level speculative execution, you must include the `speculation.h` file in your source code.

The built-in functions for thread-level speculative execution have no effect unless the thread-level speculative execution is enabled with the "-qsmp=speculative" compiler option.

Structure of statistic counters

The following structure that contains the statistic counters can be used with the thread-level speculative execution built-in functions. It is declared in the `speculation.h` file.

```
typedef struct SeReport_s {

    // Total number of chunks that are committed by nonespeculative threads
    unsigned long totalNONSpecCommitted;

    // Total number of chunks that are committed by speculative threads
```

```
    unsigned long totalSpecCommitted;

    // Total number of rollbacks for speculative threads
    unsigned long totalRollbacks;

    // Total number of serialization caused by JMV conflicts
    unsigned long totalSerializedJMV;

    // Total number of serialization caused by SE_MAX_NUM_ROLLBACK reached
    unsigned long totalSerializedMAXRB;

    // Total number of serialization caused by incorrect hardware
    // speculative mode or nesting
    unsigned long totalSerializedOTHER;

} SeReport_t;
```

se_get_all_stats

Purpose

With the `se_get_all_stats` built-in function, you can retrieve statistical information for the thread-level speculative execution of all threads in a program.

Prototype

```
void se_get_all_stats (SeReport_s *stats)
```

Usage

When the `se_get_all_stats` function is called, it updates the argument with cumulative statistics of all the regions of thread-level speculative execution that all hardware threads have run.

To use the `se_get_all_stats` function, you must set the SE_REPORT_STAT_ENABLE environment variable to YES.

You must use this function outside of parallel regions.

Related information

- Environment variables for thread-level speculative execution

se_print_stats

Purpose

With the `se_print_stats` built-in function, you can write statistics about thread-level speculative execution to a log file.

Prototype

```
void se_print_stats(void)
```

Example

```
...
#pragma speculative for
  for-loop

  se_print_stats()
...
```

Usage

After a region of thread-level speculative execution, you can call the `se_print_stats` function to write cumulative statistics of all the regions of thread-level speculative execution that each hardware thread has run to a log file.

To enable the statistics log file to be generated at each call to the `se_print_stats` function, you must set the SE_REPORT_LOG environment variable to FUNC, ALL, or VERBOSE.

By default, the log file is named `se_report.log.`*rank* where *rank* is the MPI rank of the process that called the `se_print_stats` function. The file is saved in the current working directory of the program that uses thread-level speculative execution. To override the defaults, you can specify the SE_REPORT_NAME environment variable.

You must use this function outside of parallel regions.

Note: If you use this function in a region of thread-level speculative execution, the code of that region is run in irrevocable mode.

Related information

- Environment variables for thread-level speculative execution

reset_speculation_mode

Purpose

With the `reset_speculation_mode` built-in function, you can switch mode in between thread-level speculative execution (SE) and transactional memory (TM) at runtime.

Prototype

```
void reset_speculation_mode(void)
```

Example

```
...
#pragma speculative for
  for-loop

  reset_speculation_mode()

#pragma tm_atomic
  code-block
...
```

Usage

The call to the `reset_speculation_mode` function is needed only when you are switching mode between TM and SE.

If your program consists of both TM and SE regions, it is recommended that you make a call to the `reset_speculation_mode` function; otherwise, undefined behavior might be caused.

When you call the `reset_speculation_mode` function, make sure all the TM or SE executions before the call are completed.

Related information

- "Nesting OpenMP, transactional memory, and thread-level speculative execution" on page 355

Built-in functions for transactional memory

To call the built-in functions that are specific to transactional memory, you must include the `speculation.h` file in your source code.

The built-in functions for transactional memory have no effect unless transactional memory is enabled with the "-qtm" compiler option.

Structure of statistic counters

The following structure that contains the statistic counters can be used with the transactional memory built-in functions. It is declared in the `speculation.h` file.

```
typedef struct TmReport_s {

    // Thread ID
    unsigned long hwThreadId;

    // Total number of transactions
    unsigned long totalTransactions;

    // Total number of rollbacks for transactional memory threads
    unsigned long totalRollbacks;

    // Total number of serialization caused by JMV conflicts
    unsigned long totalSerializedJMV;

    // Total number of serialization caused by TM_MAX_NUM_ROLLBACK reached
    unsigned long totalSerializedMAXRB;

    // Total number of serialization caused by incorrect hardware
    // speculative mode or nesting
    unsigned long totalSerializedOTHER;

} TmReport_t;
```

tm_get_stats
Purpose

With the `tm_get_stats` built-in function, you can retrieve statistical information for the transactional memory of a particular hardware thread in your program.

Prototype

```
void tm_get_stats (TmReport_t *stats)
```

Usage

When the `tm_get_stats` built-in function is called, it updates the argument with cumulative statistics of all the transactional atomic regions that a particular hardware thread has run.

To use the `tm_get_stats` function, you must set the TM_REPORT_STAT_ENABLE environment variable to YES.

Related information

- Environment variables for transactional memory

tm_get_all_stats

Purpose

With the `tm_get_all_stats` built-in function, you can retrieve statistical information for the transactional memory of all hardware threads in a program.

Prototype

```
void tm_get_all_stats (TmReport_t *stats)
```

Usage

When the `tm_get_all_stats` built-in function is called, it updates the argument with cumulative statistics of all the transactional atomic regions that all hardware threads have run.

To use the `tm_get_all_stats` function, you must set the TM_REPORT_STAT_ENABLE environment variable to YES.

You must use this built-in function outside of parallel regions.

Related information

- Environment variables for transactional memory

tm_print_stats

Purpose

With the `tm_print_stats` built-in function, you can write statistics for the transactional memory of a particular hardware thread to a log file.

Prototype

```
void tm_print_stats(void)
```

Example

```
  ...
#pragma tm_atomic
  code-block

  tm_print_stats()
  ...
```

Usage

After a transactional atomic region, you can call the `tm_print_stats` built-in function to write cumulative statistics of all the transactional atomic regions that a particular hardware thread has run to a log file.

To enable the statistics log file to be generated at each call to the `tm_print_stats` function, you must set the TM_REPORT_LOG variable to FUNC, ALL, or VERBOSE.

By default, the log file is named `tm_report.log.`*rank*, where *rank* in the extension is the MPI rank of the process that called `tm_print_stats`. The log file is saved in the current working directory of the program that uses transactional memory. To override the defaults, you can specify the TM_REPORT_NAME environment variable.

Note: If you use this function in a transactional atomic region, it causes the transaction running in irrevocable mode.

Related information

- Environment variables for transactional memory

tm_print_all_stats

Purpose

With the `tm_print_all_stats` built-in function, you can write statistical information for the transactional memory of all hardware threads in a program to a log file.

Prototype

```
void tm_print_all_stats(void)
```

Example

```
  ...
#pragma tm_atomic
  code-block

  tm_print_all_stats()
  ...
```

Usage

When the `tm_print_all_stats` built-in function is called, it writes the cumulative statistics of all the transactional atomic regions that all hardware threads have run to a log file.

By default, the log file is named `tm_report.log.`*rank*, where *rank* in the extension is the MPI rank of the process that called `tm_print_all_stats`. The log file is saved in the current working directory of the program that uses transactional memory. To override the defaults, you can specify the TM_REPORT_NAME environment variable.

To enable the statistics log file to be generated at each call to the `tm_print_all_stats` function, you must set the TM_REPORT_LOG variable to FUNC, ALL, or VERBOSE.

You must use this function outside of parallel regions.

Related information

- Environment variables for transactional memory

reset_speculation_mode

Purpose

With the `reset_speculation_mode` built-in function, you can switch mode in between thread-level speculative execution (SE) and transactional memory (TM) at runtime.

Prototype

```
void reset_speculation_mode(void)
```

Example

```
...
#pragma speculative for
  for-loop

  reset_speculation_mode()

#pragma tm_atomic
  code-block
...
```

Usage

The call to the `reset_speculation_mode` function is needed only when you are switching mode between TM and SE.

If your program consists of both TM and SE regions, it is recommended that you make a call to the `reset_speculation_mode` function; otherwise, undefined behavior might be caused.

When you call the `reset_speculation_mode` function, make sure all the TM or SE executions before the call are completed.

Related information

- "Nesting OpenMP, transactional memory, and thread-level speculative execution" on page 355

Chapter 7. Runtime functions for parallel processing

OpenMP runtime 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 Application Program Interface specification in www.openmp.org.

Related information

- "Environment variables for parallel processing" on page 23

omp_get_max_active_levels

Purpose

Retrieves the value of the *max-active-levels-var* internal control variable that determines the maximum number of nested active parallel regions. *max-active-levels-var* can be set with the *OMP_MAX_ACTIVE_LEVELS* environment variable or the omp_set_max_active_levels function.

Prototype

```
int omp_get_max_active_levels(void);
```

omp_set_max_active_levels

Purpose

Sets the value of the *max-active-levels-var* internal control variable to the value in the argument. If the number of parallel levels requested exceeds the number of the supported level of parallelism, the value of *max-active-levels-var* is set to the number of parallel levels supported by the runtime. If the number of parallel levels requested is not a positive integer, this routine call is ignored. When the nested parallelism is turned off, this routine has no effects and the value of *max-active-levels-var* remains 1. *max-active-levels-var* can also be set with the *OMP_MAX_ACTIVE_LEVELS* environment variable. To retrieve the value for *max-active-levels-var*, use the *omp_get_max_active_levels* function.

Prototype

```
void omp_set_max_active_levels(int max_levels);
```

omp_get_schedule

Purpose

Returns the *run-sched-var* internal control variable of the team that is processing the parallel region. The argument *kind* returns the type of schedule that will be used. *modifier* represents the chunk size that is set for applicable schedule types. *run-sched-var* can be set with the *OMP_SCHEDULE* environment variable or the **omp_set_schedule** function.

Prototype

```
int omp_get_schedule(omp_sched_t * kind, int * modifier );
```

Parameters

kind
The value returned for *kind* is one of the schedule types affinity, auto, dynamic, guided, runtime, or static.

modifier
For the schedule type dynamic, guided, or static, modifier is the chunk size that is set. For the schedule type auto, modifier has no meaning.

Related reference:

"omp_set_schedule"

Related information:

"OMP_SCHEDULE" on page 32

omp_set_schedule

Purpose

Sets the value of the *run-sched-var* internal control variable. Use **omp_set_schedule** if you want to set the schedule type separately from the *OMP_SCHEDULE* environment variable.

Prototype

```
void omp_set_schedule (omp_sched_t kind, int modifier);
```

Parameters

kind
Must be one of the schedule types affinity, auto, dynamic, guided, runtime, or static.

modifier
For the schedule type dynamic, guided, orstatic, modifier is the chunk size that you want to set. Generally it is a positive integer. If the value is less than one, the default will be used. For the schedule type auto, modifier has no meaning.

Related reference:

"omp_get_schedule" on page 499

Related information:

"OMP_SCHEDULE" on page 32

omp_get_thread_limit

Purpose

Retrieves the maximum number of OpenMP threads stored in the *thread-limit-var* internal control variable that are available to the program. *thread-limit-var* can be set with the *OMP_THREAD_LIMIT* environment variable.

Prototype

```
int omp_get_thread_limit(void);
```

omp_get_level

Purpose

Use **omp_get_level** to return the number of active and inactive nested parallel regions that the generating task is executing in. This does not include the implicit parallel region.

Prototype

```
int omp_get_level(void);
```

omp_get_ancestor_thread_num

Purpose

Use **omp_get_ancestor_thread_num** to return the thread number in the current level of the ancestor that is at the specified nested level.
omp_get_ancestor_thread_num returns -1 if the nested level is not within the range of 0 and the current thread's nested level as returned by **omp_get_level**.

Prototype

```
int omp_get_ancestor_thread_num(int level);
```

omp_get_team_size

Purpose

Use **omp_get_team_size** to return the thread team size that the ancestor belongs to.
omp_get_team_size returns -1 if the nested level is not within the range of 0 and the current thread's nested level as returned by **omp_get_level**.

Prototype

```
int omp_get_team_size(int level);
```

omp_get_active_level

Purpose

Use **omp_get_active_level** to return the number of active parallel regions that are nested.

Prototype

```
int omp_get_active_level(void);
```

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 for a subsequent parallel region by setting the first value of *num_list* for OMP_NUM_THREADS.

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 first value of *num_list* for the OMP_NUM_THREADS environment variable. This value is the maximum number of threads that can be used to form a new team if a parallel region without a **num_threads** clause is encountered.

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_final

Purpose

Returns a nonzero integer value if the function is called in a final task region; otherwise, it returns 0.

Prototype

int omp_in_final(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.

POMP callback functions

This section describes POMP callback functions that are supported by the SMP runtime on Blue Gene/Q platforms.

To use POMP, you must provide implementation of all these callback functions by building a timer probe library. For details, see Timer probe library.

Note: Nested parallelism is not supported by POMP.

POMP_Init

Purpose

This function is called when the SMP runtime is initialized. It is invoked from the master OpenMP thread.

Prototype

```
int32_t POMP_Init(void);
```

Usage

Regardless of whether the target program uses OpenMP, `POMP_Init` is always called by the POMP-enabled SMP runtime.

The `int32_t` type is defined in the `stdint.h` file.

POMP_Finalize

Purpose

This function is called when the SMP runtime is finalized. It is invoked from the master OpenMP thread.

Prototype

```
int32_t POMP_Finalize(void);
```

Usage

`POMP_Finalize` is called after all other POMP callback functions have completed.

The `int32_t` type is defined in the `stdint.h` file.

POMP_Get_handle

Purpose

This function is called whenever the SMP runtime is entering a new parallel region or loop construct. It is invoked from the master OpenMP thread.

It parses the CTC string and places meaningful location information into the POMP handle. The POMP handle is passed to all other POMP callback functions in the parallel region or loop construct. `POMP_Get_handle` exists so that the parsing can be completed once with minimal overhead for future callbacks.

Prototype

```
int32_t POMP_Get_handle(POMP_Handle_t *handle, char ctc[]);
```

Parameters

ctc[]

CTC string, used to encode source locations.

A typical CTC string consists of several fields separated by "*". It starts with an integer that indicates the length of the string, excluding the length of the integer itself, and ends with "**".

```
43*rtype=loop*sscl=test.c:19*escl=test.c:19*
```

In this example, `rtype` is the source type (`loop` or `pregion`); `sscl` is the starting source column; `escl` is the ending source column and line. The whole string indicates a loop construct in `test.c`, which begins on Line 19 and ends on Line 19.

***handle**

POMP handle, allocated by the SMP runtime with enough memory to store an 8-byte pointer. The timer probe library also allocates memory and sets the 8-byte pointer (allocated by the runtime) to point to this memory.

The type of `POMP_Handle_t` is `void*`. so the type of ***handle** is `void**`.

In the sample timer probe library, the memory allocated is for storing the parsed CTC string. To avoid memory leak, you must carefully manage the memory allocation for both the SMP runtime and the timer probe library.

Usage

Under the POMP-enabled SMP runtime, `POMP_Get_handle` is called right before `POMP_Parallel_begin` or `POMP_Loop_chunk_begin`. The memory for `POMP_Handle_t` is leaked. If you encounter issues, use static variables to store parsed CTC strings, so that only the 8 bytes (on a 64-bit system) that the runtime allocates is leaked.

No matter whether POMP is used, the CTC strings themselves are located in the static data segment of the compiled program, so it is safe to maintain pointers within the CTC strings. As long as you do not modify the strings, this can be used to, for example, set ***handle** to point to the beginning of the `sscl` filename.

Related information

- Sample timer probe library
- "POMP_Parallel_begin" on page 508
- "POMP_Loop_chunk_begin" on page 510

POMP_Parallel_enter

Purpose

This function is called when the SMP runtime is entering a parallel region. It is invoked from the master OpenMP thread.

Prototype

```
void POMP_Parallel_enter(POMP_Handle_t *handle,
                         int32_t thread_id,
                         int32_t num_threads,
                         int32_t if_expr_result,
                         char ctc[]);
```

Parameters

ctc[]
CTC string. For details, see "POMP_Get_handle" on page 506.

***handle**
POMP handle. For details, see "POMP_Get_handle" on page 506.

thread_id
The ID of the thread that enters the parallel region.

num_threads
Number of threads.

if_expr_result
The result of the `if` expression.

Usage

The `int32_t` type is defined in the `stdint.h` file.

POMP_Parallel_exit

Purpose

This function is called when the SMP runtime is exiting a parallel region. It is invoked from the master OpenMP thread.

Prototype

```
int32_t POMP_Parallel_exit(POMP_Handle_t *handle, int32_t thread_id);
```

Parameters

***handle**
POMP handle. For details, see "POMP_Get_handle" on page 506.

thread_id
The ID of the thread that exits the parallel region.

Usage

The `int32_t` type is defined in the `stdint.h` file.

POMP_Parallel_begin

Purpose

This function is called when the SMP runtime begins execution in a parallel region. It is invoked from all OpenMP threads.

Prototype

```
int32_t POMP_Parallel_begin(POMP_Handle_t *handle, int32_t thread_id);
```

Parameters

***handle**
POMP handle. For details, see "POMP_Get_handle" on page 506.

thread_id
The ID of the thread that begins its execution in the parallel region.

Usage

The `int32_t` type is defined in the `stdint.h` file.

POMP_Parallel_end

Purpose

This function is called when the SMP runtime ends execution in a parallel region. It is invoked from all OpenMP threads.

Prototype

```
int32_t POMP_Parallel_end(POMP_Handle_t *handle, int32_t thread_id);
```

Parameters

`*handle`
POMP handle. For details, see "POMP_Get_handle" on page 506.

`thread_id`
The ID of the thread that ends its execution in the parallel region.

Usage

The `int32_t` type is defined in the `stdint.h` file.

POMP_Loop_enter

Purpose

This function is called when the SMP runtime is entering a parallel loop construct. It is invoked from the master OpenMP thread.

Prototype

```
int32_t POMP_Loop_enter(POMP_Handle_t *handle,
                        int32_t thread_id,
                        int64_t chunk_size,
                        int64_t init_iter,
                        int64_t final_iter,
                        int64_t incr,
                        char ctc[]);
```

Parameters

`ctc[]`
CTC string. For details, see "POMP_Get_handle" on page 506.

`*handle`
POMP handle. For details, see "POMP_Get_handle" on page 506.

`thread_id`
The ID of the thread that enters the parallel loop construct.

`chunk_size`
The value of **chunk_size** is reliable only for static work distribution. When dynamic chunking or other schemes are used, **chunk_size** might have a random value, or most probably, zero.

`init_iter`
The initial iteration of the loop.

final_iter
The final iteration of the loop.

incr
The increment of the loop, and is 1 each time.

Usage

The `int32_t` and `int64_t` types are defined in the `stdint.h` file.

POMP_Loop_exit

Purpose

This function is called when the SMP runtime is exiting a parallel loop construct. It is invoked from the master OpenMP thread.

Prototype

```
int32_t POMP_Loop_exit(POMP_Handle_t *handle, int32_t thread_id);
```

Parameters

***handle**
POMP handle. For details, see "POMP_Get_handle" on page 506.

thread_id
The ID of the thread that exits the parallel loop construct.

Usage

The `int32_t` type is defined in the `stdint.h` file.

POMP_Loop_chunk_begin

Purpose

This function is called when the SMP runtime begins the chunk of iterations of a parallel loop construct. It is invoked from all OpenMP threads.

Prototype

```
int32_t POMP_Loop_chunk_begin(POMP_Handle_t *handle,
                              int32_t thread_id,
                              int64_t init_iter,
                              int64_t final_iter);
```

Parameters

***handle**
POMP handle. For details, see "POMP_Get_handle" on page 506.

thread_id
The ID of the thread that enters the chunk of iterations.

init_iter
The initial iteration of the loop.

final_iter
The final iteration of the loop.

Usage

`POMP_Loop_chunk_begin` is invoked very frequently, so keep the processing of timer probe library in it to a minimum.

The `int32_t` and `int64_t` types are defined in the `stdint.h` file.

POMP_Loop_chunk_end

Purpose

This function is called when the SMP runtime ends the chunk of iterations of a parallel loop construct. It is invoked from all OpenMP threads.

Prototype

```
int32_t POMP_Loop_chunk_end(POMP_Handle_t *handle, int32_t thread_id);
```

Parameters

`*handle`
POMP handle. For details, see "POMP_Get_handle" on page 506.

`thread_id`
The ID of the thread that completes the chunk of iterations.

Usage

`POMP_Loop_chunk_end` is invoked very frequently, so keep the processing of timer probe library in it to a minimum.

The `int32_t` type is defined in the `stdint.h` file.

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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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, 2010.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" 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, other countries, or both.

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

A

B

C

D

E

F

H

W

X

Product Number: 5799-AG1

Printed in USA

GC14-7363-00